AD-A054 943 SYRACUSE UNIV N Y F/G 9/2
LARGE SCALE INFORMATION SYSTEMS. VOLUME II.(U)
MAR 78 F30602-74-C-0335
UNCLASSIFIED RADC-TR-78-43-VOL-2 NL

1 OF 3
AD
A054943

FOR FURTHER TRAN

AD A054943

DDC FILE COPY

RADC-TR-78-43, Volume II (of four)
Final Technical Report
March 1978

LARGE SCALE INFORMATION SYSTEMS

Syracuse University

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-43, Volume II (of four) has been reviewed and is approved for publication.

APPROVED: James L. Previte

JAMES L. PREVITE
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: RADC-TR-78-43, Vol II (of Four Vols) (2)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): LARGE SCALE INFORMATION SYSTEMS, Volume II.

(9) 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. 3 Jun 74 — 2 Feb 77.

6. PERFORMING ORG. REPORT NUMBER: N/A

7. AUTHOR(s): Syracuse University

(15) 8. CONTRACT OR GRANT NUMBER(s): F30602-74-C-0335

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Syracuse University, Syracuse New York 13210

(16) 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810244 (17) 02

11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCA), Griffiss AFB NY 13441

(11) 12. REPORT DATE: March 1978

13. NUMBER OF PAGES: 214

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 204 p.

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
James L. Previte (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel Processing
Programming Languages
Simulation
Data Base Management
Computer Architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes and references work conducted by Syracuse University in four broad areas: parallel processing, programming languages, modeling, and performance evaluation of generalized data management systems.

A number of applications were evaluated for processing by an associateive processor architecture including air traffic control, carryless arithmetic and simulation of high-speed random logic. An extension of the typed lambda (Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339600

calculus has been developed which permits the binding and application of types. User-defined types and procedural data structures are shown to be complementary tools for data abstraction. Direct and continuation semantics of the domain of flow diagrams are formulated and the properties explored. The use of transition diagrams as a tool for structured programming has been investigated. A variety of concepts and notations have been devised to facilitate reasoning about arrays.

Work relation to various simulation tasks are reported on. A tutorial on the current statistical methods of analyzing simulation output data is provided.

A number of tasks relating to file systems are discussed and a framework is advanced for describing various file organizations and operations on files.

Preface

This report describes efforts completed in the Large Scale Information Systems project at Syracuse University under RADC contract F30602-74-C-0335. The work covers the period June 3, 1974 through February 2, 1977.

The report is produced in four volumes to facilitate single volume distribution.

Contents of Volume 1

Contents of Volume 2

Contents of Volume 3

Contents of Volume 4

Section 8

REASONING ABOUT ARRAYS*

John C. Reynolds

Syracuse University

ABSTRACT

A variety of concepts, laws, and notations are presented which facilitate reasoning about arrays. The basic concepts include intervals and their partitions, functional restriction, images, pointwise extension of relations, ordering, single-point variation of functions, various equivalence relations for array values, and concatenation. The effectiveness of these ideas is illustrated by informal descriptions of algorithms for binary search and merging, and by a short formal proof.

*Work supported by National Science Foundation Grant MCS 75-22002.

1. Introduction

The use of assertions to describe programs and prove their correctness[1,2,3] has developed to the point where the necessary assertions are often at least as lengthy and difficult to comprehend as the program which they describe. A major cause is the use of languages and proof methods - typically the first-order predicate calculus - which are taken from classical logic and are not oriented towards programming.

Perhaps the most glaring example of these difficulties is the use of arrays. One need only compare the assertions needed to describe a program such as n log n exponentiation, which does not involve arrays or other compound data structures, with the assertions for a program such as binary search, which is intuitively no more complex, but uses arrays. In the first case, the assertions are clear and concise, and reasoning about them involves only the familiar laws of elementary algebra. But when arrays are introduced, the assertions become lengthy and filled with quantifiers, and their manipulation seems only tenuously connected with the programmer's intuition.

Superficially, we need a better notation for assertions about arrays. But more fundamentally, we need concepts and laws which are not only correct but also reflect our intuitive understanding of arrays, just as the concepts of addition and multiplication, and the associative, commutative, and distributive laws reflect our intuitive understanding of numbers. Once the right concepts and laws have been found, it is comparatively trivial to design a notation which facilitates their application.

This paper presents a variety of concepts, laws, and notations for reasoning about arrays - some borrowed from mathematics and others original - which we believe meet the above criteria. Their utility will be demonstrated both by informal descriptions of program behavior and by a short formal proof of program correctness.

For a programming language, we will use Algol 60 with the following changes:

(1) while statements.

(2) Round rather than square brackets for array subscripts (which emphasizes the view that array values are functions).

(3) Integer expressions of the form lower X and upper X, denoting the minimum and maximum subscripts of a one-dimensional array X.

(Although we will not use procedures here, it should be noted that for (3) to be fully useful, there must be some way of restricting the interval of subscripts of an actual array parameter.)

We have purposely stayed close to Algol to avoid inadvertently choosing a programming language which hid the defects of our assertion language. In particular, we have refrained from introducing our notation for assertions into the programming language itself (except for lower and upper, which were irresistably attractive). Moving in this direction seems to lead to a very high level language, closer to APL than to Algol, which is beyond the scope of this paper.

On the other hand, even the choice of Algol has had subtle effects on the ensuing development. For example, switching to a programming language with the novel approach to arrays described in Chapter 11 of Reference 4 would necessitate minor changes to many concepts, such as abandoning the uniqueness of the array value with an empty domain.

To an even greater extent than is indicated by the explicit references, this work is built upon the ideas of C. A. R. Hoare.[5,6,7] Mention should also be made of distinct but related work on arrays by D. C. Cooper[8] and of work by R. Burstall[9] which, roughly speaking, does for list structures what we are trying to do for arrays.

2. Interval and Partition Diagrams

Before considering arrays themselves, we introduce some diagrammatic expressions for making assertions about subscripts. Basically, these expressions are a formalization of the diagrams which are traditionally drawn by programmers when describing arrays.

An interval is a finite consecutive set of integers. If a and b are expressions denoting integers, then a⌈ b⌉, called an interval diagram, is an expression denoting the interval

$$a\lceil\ b\rceil \equiv \{i \mid a < i \leq b\} \ .$$

When formulating general properties of interval diagrams (or partition diagrams) we will always use the standard form a⌈ b⌉. But when using the diagrams to make assertions, we will permit more flexibility.

Specifically, at either end of an interval diagram, |a may be written instead of a-1|. Also, [a] may be written as an abbreviation for [a a]. Thus

$$[a\ \ b] = \{i \mid a \le i \le b\}$$
$$[a\ \ )b = \{i \mid a \le i < b\}$$
$$a(\ \ )b = \{i \mid a < i < b\}$$
$$[a] = \{a\} \ .$$

For any finite set S, we write #S to denote the size, or number of elements in S. Thus

$$\#\, a[\ \ b] = \underline{\text{if}}\ b - a \ge 0\ \underline{\text{then}}\ b - a\ \underline{\text{else}}\ 0 \ . \qquad (2.1)$$

This use of a conditional expression to describe a fundamental property of a data structure is a clear symptom of a potential source of error, i.e., the possibility that a program may be correct for one case of the conditional but not the other. To emphasize this situation, we say that the interval a[b] is <u>regular</u> when b - a ≥ 0, or <u>irregular</u> when b - a < 0. It is evident that a nonempty interval is always regular, but the empty interval can be either regular or irregular. (This is a slight abuse of language; it is really the interval diagram, rather than the interval itself, which is regular or irregular.)

From interval diagrams, we can build more complex entities called partition diagrams, which describe relationships between intervals. If $a_0, a_1, \ldots, a_n$ are expressions denoting integers, then:

(a) $a_0[\ a_1\ |\ \cdots\ a_{n-1}\ |\ a_n\ ]$ is called a <u>partition diagram</u>.

(b) $a_0[\ a_1\ ]$, ... , $a_{n-1}[\ a_n\ ]$, i.e. the intervals denoted by diagrams obtained by eliminating all but an adjacent pair of lines, are called the <u>component intervals</u> of the partition diagram.

(c) $a_0[\ a_n\ ]$, i.e. the interval denoted by the diagram obtained by eliminating interior lines, is called the <u>total interval</u> of the partition diagram.

(d) The partition diagram is a logical expression which is true iff the component intervals are a partition of the total interval, i.e., iff the component intervals are disjoint and their union is the total interval.

As with interval diagrams, ⋯ |a ⋯ may be written in place of ⋯ a-1| ⋯, and ⋯ |a| ⋯ in place of ⋯ |a a| ⋯. Thus for example, |a |b| c] is a partition diagram which is true iff the component intervals [a b = {i | a ≤ i < b}, [b] = {b}, and b[c] = {i | b < i ≤ c} are disjoint and their union is the total interval [a c] = {i | a ≤ i ≤ c}.

The nature of partitions implies that the size of the total interval is the sum of the sizes of the component intervals:

$$a_0[\ a_1\ |\ \cdots\ a_{n-1}\ |\ a_n\ \text{implies}\ \#\, a_0[\ a_n] = \sum_{i=1}^{n} \#\, a_{i-1}[\ a_i] \quad (2.2)$$

As shown in the Appendix, (2.2) implies the following fundamental property of partition diagrams:

$$a_0[\ a_1\ |\ \cdots\ a_{n-1}\ |\ a_n\ \text{iff either} \quad (2.3)$$

$$a_0 \le a_1 \le \cdots \le a_{n-1} \le a_n \quad \text{or} \quad a_0 \ge a_1 \ge \cdots \ge a_{n-1} \ge a_n .$$

Note that the first inequality asserts that every component interval is regular, while the second inequality asserts that every component interval is empty.

From (2.3), the following simple cases are obvious:

a[b] is always true. (2.4)

[a b] iff [a b] iff a ≤ b iff [a b] is nonempty. (2.5)

[a |b| c] iff a ≤ b ≤ c iff b ∈ [a c]. (2.6)

More interestingly, one can easily derive several "diagrammatically natural" rules of inference: (Here "line" refers to any vertical line in a diagram, including its associated expression.)

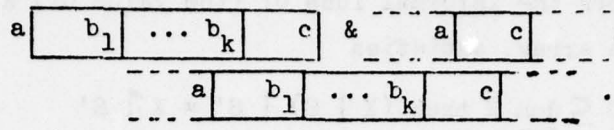
Erasure From a partition diagram one can infer any diagram obtained by deleting a line, i.e., (2.7)

⋯ |a ⋯ implies ⋯ ⋯ .

Adjacent Duplication From a partition diagram one can infer any diagram obtained by replicating a line next to itself, i.e., (2.8)

⋯ |a ⋯ implies ⋯ |a a| ⋯ .

Substitution From two partition diagrams such that the end lines of the first match some pair of adjacent lines in the second, one can infer the diagram obtained by substituting the first diagram for the adjacent lines in the second: (2.9)



The use of these rules is illustrated by the following inferences, which will be pertinent to the binary search example given later:

(a) For any integers ℓ and u, (2.4) and (2.8) show that $\boxed{\ell \;\; \ell \;\; u \;\; u}$ holds.

(b) Suppose $\boxed{\ell \;\; a \;\; b \;\; u}$ and $a \leq j \leq b$. Then by (2.6) and (2.9), $\boxed{\ell \;\; a \;\; j \;\; b \;\; u}$ holds. In turn, by (2.7), this implies $\boxed{\ell \;\; j \;\; u}$, $\boxed{\ell \;\; j+1 \;\; b \;\; u}$, and $\boxed{\ell \;\; a \;\; j-1 \;\; u}$.

3. Functions as Array Values

There are two quite different concepts of an array. The more traditional view is that an array of, say, real numbers is a function from subscripts into variables, which in turn possess real values. The more recent view, expounded by Hoare[5,6] and Dijkstra[4], is that an array of real numbers is a variable whose value is a function from subscripts into real numbers. In this paper, we take the latter view. The effect is to banish the possibility of "sharing" or "aliasing" among array elements, which would greatly complicate the problems of proving program correctness.

Specifically, we assume that an array declared by τ array X(a:b) is a variable whose values range over the set of functions from the interval $\boxed{a \;\; b}$ into the set τ.

For any function X, we write dom X for the domain of X and, when this domain is an interval, lower X and upper X for the integers such dom X = $\boxed{\text{lower X} \;\; \text{upper X}}$. This definition of lower and upper is intentionally incomplete for the case where X is the unique function, denoted by <>, whose domain is the empty set.

When $S \subseteq \text{dom } X$, we write $X \upharpoonright S$, called the restriction of X to S, to denote the function such that

$$\text{dom}(X \upharpoonright S) = S \tag{3.1}$$

$$(\forall i \in S)\ (X \upharpoonright S)(i) = X(i)\ . \tag{3.2}$$

This concept, which mirrors the informal idea of (the value of) a subarray or segment of an array, satisfies

$$\text{If } S' \subseteq S \subseteq \text{dom } X \text{ then } (X \upharpoonright S) \upharpoonright S' = X \upharpoonright S' \tag{3.3}$$

$$X \upharpoonright \{\} = \langle\rangle\ . \tag{3.4}$$

As an example, consider the program

```
begin integer i; integer array Squares(-5: 5);
integer array Possquares(0: 5);
integer array Nosquares(14: 5);
for i := -5 until 5 do Squares(i) := i × i;
for i := 0 until 5 do Possquares(i) := i × i;
...
end
```

At the program point indicated by the ellipsis, the following assertions will hold:

$$\text{dom Squares} = \boxed{-5 \quad 5}$$

$$\text{lower Squares} = -5$$

$$\text{upper Squares} = 5$$

$$(\forall i \in \boxed{-5 \quad 5}\ )\ \text{Squares}(i) = i \times i$$

$$\text{Possquares} = \text{Squares} \upharpoonright \boxed{0 \quad 5}$$

$$\text{Nosquares} = \text{Squares} \upharpoonright \{\} = \langle\rangle$$

$$\text{lower Nosquares} > \text{upper Nosquares}\ .$$

The expressions lower X and upper X occur so frequently in interval and partition diagrams that it is useful to adopt conventions for eliding them unambiguously. When an interval or partition diagram is labelled with a function X, lower X may be omitted from the right of the leftmost line of the diagram, and upper X may be omitted from the left of the rightmost line. For example, X: $\boxed{\quad|\ a \quad b\ |\quad}$ stands for $\boxed{\text{lower X}\ |\ a \quad b\ |\ \text{upper X}}$. Moreover, when an interval diagram is used to restrict a function X, the label can also be elided. For example, $X \upharpoonright \boxed{\quad a}$ stands for $X \upharpoonright \boxed{\text{lower X} \quad a}$.

For a function X, we write {X}, called the image of X, to denote the set $\{X(i) \mid i \in \underset{\sim}{\mathrm{dom}}\ X\}$ of values obtained by applying X to members of its domain. (On the other hand, when x is not a function, {x} will denote the singleton set containing x.) Thus for example,

$$\{\underline{\text{Possquares}}\} = \{0, 1, 4, 9, 16, 25\}$$
$$\{\underline{\text{Possquares}} \uparrow [1 \quad 3]\} = \{1, 4, 9\}$$
$$\{\underline{\text{Squares}} \uparrow [-2 \quad 2]\} = \{0, 1, 4\} \ .$$

It is easily seen that images possess the following properties:

$$S \subseteq \underset{\sim}{\mathrm{dom}}\ X \text{ implies } \{X \uparrow S\} \subseteq \{X\} \tag{3.5}$$
$$\{<>\} = \{\} \tag{3.6}$$
$$S \cup S' = \underset{\sim}{\mathrm{dom}}\ X \text{ implies } \{X\} = \{X \uparrow S\} \cup \{X \uparrow S'\} \tag{3.7}$$
$$\{X \uparrow [i]\} = \{X(i)\} \tag{3.8}$$
$$\#\{X\} \leq \#\underset{\sim}{\mathrm{dom}}\ X \ . \tag{3.9}$$

4. Operations on Relations

Suppose ρ is a binary relation between two sets U and U'. Then ρ^*, called the pointwise extension of ρ, is the binary relation between the set of subsets of U and the set of subsets of U', such that $S\ \rho^*\ S'$ holds if and only if $x\ \rho\ x'$ holds for all x in S and x' in S'.

When U and U' are both the set of integers, ρ could be any of the relational operators of Algol. For example, $\{2, 3\} \leq^* \{3, 4\}$ and $\{2, 3\} \neq^* \{4, 5\}$ are both true, while $\{2, 3\} <^* \{3, 4\}$, $\{2, 3\} =^* \{2, 3\}$, and $\{2, 3\} \neq^* \{2, 3\}$ are all false. The last two examples demonstrate that $\neq^*$ is not the negation of $=^*$ (and thereby show the importance of making * explicit).

The pointwise extension of any relation satisfies the following laws:

$$S\ \rho^*\ S' \ \&\ T \subseteq S \text{ implies } T\ \rho^*\ S' \tag{4.1a}$$
$$S\ \rho^*\ S' \ \&\ T' \subseteq S' \text{ implies } S\ \rho^*\ T' \tag{4.1b}$$
$$\{\}\ \rho^*\ S' \tag{4.2a}$$
$$S\ \rho^*\ \{\} \tag{4.2b}$$
$$(S \cup T)\ \rho^*\ S' \text{ iff } S\ \rho^*\ S' \ \&\ T\ \rho^*\ S' \tag{4.3a}$$
$$S\ \rho^*\ (S' \cup T') \text{ iff } S\ \rho^*\ S' \ \&\ S\ \rho^*\ T' \tag{4.3b}$$
$$\{x\}\ \rho^*\ \{x'\} \text{ iff } x\ \rho\ x' \ . \tag{4.4}$$

Occasionally, one needs the pointwise extension of a relation with regard to only a single argument. The simplest way of encompassing this case is to regard $x\ \rho^*\ S'$ as an abbreviation for $\{x\}\ \rho^*\ S'$ and $S\ \rho^*\ x'$ as an abbreviation for $S\ \rho^*\ \{x'\}$.

Another concept involving relations, somewhat more specialized than pointwise extension, is ordering. The usual idea of an ordered array can be generalized to an arbitrary relation in a way which unifies several important cases. Let X be a function whose domain is a set of integers, and let ρ be a binary relation appropriate to the type of result of X. Then X is ordered with regard to ρ, written $\underset{\sim}{\text{ord}}_\rho\ X$, if and only if, for all i and j in the domain of X, $i < j$ implies $X(i)\ \rho\ X(j)$.

The following "orderings" appear as specific cases:

$\underset{\sim}{\text{ord}}_{\leq}$ X: increasing order

$\underset{\sim}{\text{ord}}_{<}$ X: strict increasing order

$\underset{\sim}{\text{ord}}_{\geq}$ X: decreasing order

$\underset{\sim}{\text{ord}}_{>}$ X: strict decreasing order

$\underset{\sim}{\text{ord}}_{=}$ X: all elements equal

$\underset{\sim}{\text{ord}}_{\neq}$ X: all elements distinct

Moreover, the generalization satisfies the following essential laws of ordering:

$$\underset{\sim}{\text{ord}}_\rho\ X\ \&\ S \subseteq \underset{\sim}{\text{dom}}\ X \text{ implies } \underset{\sim}{\text{ord}}_\rho (X \restriction S) \tag{4.5}$$

$$\not\Lambda\ \underset{\sim}{\text{dom}}\ X \leq 1 \text{ implies } \underset{\sim}{\text{ord}}_\rho\ X \tag{4.6}$$

$$\text{If } S \cup T = \underset{\sim}{\text{dom}}\ X\ \&\ S <^* T \text{ then} \tag{4.7}$$
$$(\underset{\sim}{\text{ord}}_\rho\ X \text{ iff } (\underset{\sim}{\text{ord}}_\rho (X \restriction S)\ \&\ \underset{\sim}{\text{ord}}_\rho (X \restriction T)\ \&\ \{X \restriction S\}\ \rho^*\ \{X \restriction T\}))\ .$$

An important special case of (4.7) is obtained by taking S and T to be two components of a partition:

$$\text{If X: } \boxed{\boxed{\ \ k\ }\ \ } \text{ then} \tag{4.8}$$
$$(\underset{\sim}{\text{ord}}_\rho\ X \text{ iff } (\underset{\sim}{\text{ord}}_\rho (X \restriction \boxed{\ \ k\ }\ )\ \&\ \underset{\sim}{\text{ord}}_\rho (X \restriction k\boxed{\ \ }\ )$$
$$\&\ \{X \restriction \boxed{\ \ k\ }\}\ \rho^*\ \{X \restriction k\boxed{\ \ }\}))\ .$$

5. Binary Search

We have now introduced enough of our notation to demonstrate its use in describing - precisely yet intelligibly - why a program works. As an example, we describe an algorithm for binary search.

Given an ordered array X and a test value y, the program should set the boolean variable found to indicate whether any element of X is equal to y. If found is true, then the integer variable j should be set to the subscript of X such that X(j) = y. More precisely, if ord$_{\leq}$ X, then executing the program should achieve the goal

$$\underset{\sim}{\text{if}}\ \text{found}\ \underset{\sim}{\text{then}}\ X{:}\ \boxed{\,|\,j\,|\,}\ \&\ X(j) = y\ \underset{\sim}{\text{else}}\ \{X\} \neq^{*} y\ .$$

Throughout program execution, found will only be set to true if $X{:}\ \boxed{\,|\,j\,|\,}\ \&\ X(j) = y$ is achieved. On the other hand, when found is false, it will not be known that y occurs nowhere in X, but only that it does not occur in either of two segments at the left and right ends of X. If we use the local variables a and b to delineate these segments, we have the invariant:

$$\underset{\sim}{\text{if}}\ \text{found}\ \underset{\sim}{\text{then}}\ X{:}\ \boxed{\,|\,j\,|\,}\ \&\ X(j) = y$$

$$\underset{\sim}{\text{else}}\ X{:}\ \boxed{\,|\,a\ \ b\,|\,}\ \&\ \{X \upharpoonright (\boxed{}\, a \cup b\, \boxed{})\} \neq^{*} y\ .$$

On the one hand, this invariant can be achieved initially by setting found to false and making the end segments of X empty. On the other hand, the invariant implies the goal of the program if either found is true or a > b, since the latter condition implies that $\boxed{a\ \ b}$ is empty and thus, from the partition diagram, $\boxed{}\, a \cup b\, \boxed{} = \underset{\sim}{\text{dom}}\ X$. Thus our program has the form:

```
begin integer a, b;
a := lower X; b := upper X; found := false;
while ¬(found or a > b) do ...
end .
```

When execution of the body of the while statement begins, both the invariant and the while test will be true. Since a $\leq$ b, we can perform an operation "Pick j" (whose details will be considered later) which sets j to some integer in $\boxed{a\ \ b}$. At this stage, we will have

$$X{:}\ \boxed{\,|\,a\,|\,j\,|\,b\,|\,}\ \&\ \{X \upharpoonright (\boxed{}\, a \cup b\, \boxed{})\} \neq^{*} y\ ,$$

and we can compare X(j) with y. There are three cases:

(1) If X(j) = y, the invariant will be preserved if found is set to true.

(2) If X(j) < y, then ord≤ X insures that {X↑ [j]} <* y, so that {X↑ ([j] ∪ b[ ])} ≠* y. This permits us to set a to j + 1.

(3) If X(j) > y, then a similar argument justifies setting b to j - 1.

Thus our program is:

```
begin integer a, b;
a := lower X; b := upper X; found := false;
while ¬(found or a > b) do
    begin
    "Pick j";
    if X(j) = y then found := true else
        if X(j) < y then a := j + 1 else b := j - 1
    end
end
```

Termination is guaranteed by the fact that each iteration either sets found to true, which immediately stops further iterations, or else decreases the size of [a b], whose emptiness will cause termination. The absence of subscript errors is guaranteed since X: [| j |] holds at the program points where X(j) is evaluated.

It should be noticed that this description of binary search does not exclude the possibility that [lower X upper X], and therefore [a b], might be irregular. The heart of the matter is the reasoning about partition diagrams, which was formalized at the end of Section 2. One of the virtues of this kind of reasoning is that it includes the irregular case without any special case analysis.

To complete our program, we must digress from the topic of arrays to specify "Pick j". In this case, the problem is not to find a correct realization - either j := a or j := b would be correct - but to find an efficient one. The need to shrink [a b] as much as possible suggests choosing j at or near the midpoint of [a b], i.e., j := (a + b) ÷ 2.

However, we must be sure that, if $a \leq b$, then $j := (a + b) \div 2$ will achieve $a \leq j \leq b$, despite the fact that integer division involves rounding (and that the details of this rounding might vary for different machines, especially when $a + b$ is negative). Fortunately, it is enough to know that division by two is a monotonic function which is exact for even numbers. For $a \leq b$ implies $a + a \leq a + b \leq b + b$, so that monotonicity gives $(a + a) \div 2 \leq (a + b) \div 2 \leq (b + b) \div 2$, and exactness for even numbers gives $a \leq (a + b) \div 2 \leq b$. (S. Winograd has pointed out that $j := (a + b) \div 2$ is unnecessarily prone to overflow, in comparison with, for example, $j := a + (b - a) \div 2$. We leave it to the reader to show that the correctness of his improvement can still be proved with a monotonicity argument.)

6. Array Assignment

We must now move beyond programs such as binary search which merely use arrays, to consider programs which change arrays. In programming languages at the level of Algol, the fundamental agent of change is an assignment statement which alters a single array element, e.g., $X(i) := e$.

Hoare[5,6] has shown that, to deal with this statement from the viewpoint that an array is a function-valued variable, we must regard it as an abbreviation for the assignment $X := [X \mid i \mid e]$, where $[X \mid i \mid e]$ denotes the function which is similar to X except that it maps i into e. More formally, $[X \mid i \mid e]$ is defined when $i \in \underset{\sim}{\text{dom}}\ X$, in which case it is the function satisfying

$$\underset{\sim}{\text{dom}}\,[X \mid i \mid e] = \underset{\sim}{\text{dom}}\ X \tag{6.1}$$

$$[X \mid i \mid e](i) = e \tag{6.2}$$

$$[X \mid i \mid e](j) = X(j) \quad \text{when } j \neq i\ , \tag{6.3}$$

and, as an immediate consequence of (6.3),

$$[X \mid i \mid e] \upharpoonright S = X \upharpoonright S \quad \text{when} \quad S \subseteq \text{dom } X \text{ and } i \notin S\ . \tag{6.4}$$

Once $X(i) := e$ is seen as an abbreviation for $X := [X \mid i \mid e]$, the usual axiom of assignment[2]:

$$P|_{x \to e}\ \{x := e\}\ P \tag{6.5}$$

(where $P|_{x \to e}$ denotes the result of substituting e for x in P) extends to an axiom of array assignment[6]:

$$P|_{X \to \lfloor X \mid i \mid e \rfloor}\ \{X(i) := e\}\ P \quad . \tag{6.6}$$

(Because of (6.1), when this axiom is used, the substitution $X \to \lfloor X \mid i \mid e \rfloor$ need not be applied to occurrences of X in dom X, X:, lower X, or upper X.)

7. Equivalence Relations for Arrays

For many programs which alter arrays, such as sorting programs, a full specification will stipulate both that the final value of the array will possess some property, such as being ordered, and that the final value will be related to the initial value in some way, such as being a rearrangement. Often - even when the situation is intuitively obvious - a formidable technical apparatus is needed to formulate and prove the latter kind of specification.

To deal with these problems it is useful to introduce several equivalence relations for array values. Suppose X and Y are both functions whose domains are sets of integers. Then:

(a) We write $X \approx Y$, and say that X is a redistribution of Y iff $\{X\} = \{Y\}$.

(b) We write $X \sim Y$, and say that X is a rearrangement of Y iff there is a bijection B (sometimes called a one-to-one correspondence or a permutation) from dom X to dom Y such that $(\forall i \in \text{dom } X)\ Y(B(i)) = X(i)$.

(c) We write $X \simeq Y$, and say that X is a shift of Y iff there is a bijection as in (b) with the special form $B(i) = i + s$ for some integer s.

This defines an increasingly stringent sequence of equivalence relations. Thus, where ρ is $\approx$, $\sim$, or $\simeq$:

Transitivity $X \rho Y \ \&\ Y \rho Z$ implies $X \rho Z$ (7.1)

Symmetry $X \rho Y$ implies $Y \rho X$ (7.2)

Reflexivity $X \rho X$ (7.3)

$X \simeq Y$ implies $X \sim Y$ (7.4)

$X \sim Y$ implies $X \approx Y$. (7.5)

Finally, we have three more specific laws. Exchanging a pair of elements produces a rearrangement:

$$(\forall i, j \in \underset{\sim}{\mathrm{dom}}\ X)[[X \mid i \mid X(j)] \mid j \mid X(i)] \sim X\ , \tag{7.6}$$

two one-element arrays with equal values are shifts of one another:

$$\boxed{i} = \underset{\sim}{\mathrm{dom}}\ X\ \&\ \boxed{j} = \underset{\sim}{\mathrm{dom}}\ Y\ \&\ X(i) = Y(j) \text{ implies } X \simeq Y\ , \tag{7.7}$$

and a shift of an ordered array is ordered:

$$X \simeq Y\ \&\ \underset{\sim}{\mathrm{ord}}_{\rho}\ X \text{ implies } \underset{\sim}{\mathrm{ord}}_{\rho}\ Y\ . \tag{7.8}$$

As Hoare has pointed out,[3] for any program which only alters an array by performing exchanges, (7.1), (7.3), and (7.6) are sufficient to show that the final array value is a rearrangement of the initial value. However, to deal with programs which move information from one array to another, we must also consider the concatenation of array values.

8. Concatenation

Let X and Y be functions whose domains are intervals with sizes m and n respectively. Then $X \frown Y$, called the concatenation of X and Y is the unique function such that

$$\underset{\sim}{\mathrm{dom}}\ (X \frown Y) = \boxed{1 \quad m+n}$$

$$(X \frown Y) \upharpoonright \boxed{1 \quad m} \simeq X$$

$$(X \frown Y) \upharpoonright \boxed{m+1 \quad m+n} \simeq Y\ .$$

The choice of one as a lower bound is arbitrary, since we will always regard shifts of concatenated array values as equivalent.

Let <> denote the unique function whose domain is empty. Then concatenation satisfies the following laws:

$$X \frown <> \simeq X \tag{8.1}$$

$$<> \frown X \simeq X \tag{8.2}$$

$$(X \frown Y) \frown Z \simeq X \frown (Y \frown Z) \tag{8.3}$$

$$X \simeq X'\ \&\ Y \simeq Y' \text{ implies } X \frown Y \simeq X' \frown Y' \tag{8.4}$$

$$X \frown Y \sim Y \frown X \tag{8.5}$$

$$X \sim X'\ \&\ Y \sim Y' \text{ implies } X \frown Y \sim X' \frown Y' \tag{8.6}$$

$$X\colon \boxed{\quad a \mid \quad} \text{ implies } X \simeq (X\uparrow \boxed{\quad a}) \frown (X\uparrow a\boxed{\quad}) \tag{8.7}$$

$$\{X \frown Y\} = \{X\} \cup \{Y\} \tag{8.8}$$

$$\underset{\sim}{\text{ord}}_{\rho}(X \frown Y) \text{ iff } \underset{\sim}{\text{ord}}_{\rho}\ X \ \&\ \underset{\sim}{\text{ord}}_{\rho}\ Y \ \&\ \{X\}\ \rho^{*}\ \{Y\}\ . \tag{8.9}$$

The first four laws show that array values form a monoid under concatenation, provided that shift equivalence is used in place of true equality. The next two laws show that this monoid becomes commutative when the less stringent equivalence of rearrangement is used. (Technically, one can make these statements precise by working with the quotient of the set of array values under the equivalence relations $\simeq$ or $\sim$.)

The last three laws establish the basic connections between concatenation and partitions, images, and ordering. In particular, (8.9) is a consequence of (4.8) and (7.8).

In fact (8.3) actually remains true when $\simeq$ is changed to $=$. But the stronger relationship is irrelevant, since we should never be interested in true equality for concatenated array values.

9. Merging

As a second example of program description, we consider the problem of merging: Given two ordered arrays X and Y, set Z to an ordered rearrangement of the concatenation of X and Y. We assume that Z is just the right size to hold the result. Thus if

$$\underset{\sim}{\text{ord}}_{\leq}\ X \ \&\ \underset{\sim}{\text{ord}}_{\leq}\ Y \ \&\ \#\underset{\sim}{\text{dom}}\ Z = \#\underset{\sim}{\text{dom}}\ X + \#\underset{\sim}{\text{dom}}\ Y\ ,$$

then executing the program should achieve the goal

$$\underset{\sim}{\text{ord}}_{\leq}\ Z \ \&\ Z \sim X \frown Y\ .$$

During execution, each array will be partitioned into a processed part on the left and an unprocessed part on the right, the processed part of Z will be an ordered rearrangement of the concatenation of the processed parts of X and Y, the unprocessed part of Z will be the right size to hold the unprocessed parts of X and Y, and all processed elements in Z will be smaller or equal to all unprocessed elements in X or Y. (The last condition is needed to insure that the unprocessed elements can be moved into Z without rearranging the already processed elements.) Thus we have the invariant:

$$I \equiv X: \boxed{|kx} \;\&\; Y: \boxed{|ky} \;\&\; Z: \boxed{|kz} \qquad (a)$$

$$\&\ \mathrm{ord}_{\leq}\ Z\!\uparrow\boxed{}kz \qquad (b)$$

$$\&\ Z\!\uparrow\boxed{}kz \sim X\!\uparrow\boxed{}kx \frown Y\!\uparrow\boxed{}ky \qquad (c)$$

$$\&\ \#Z: \boxed{kz} = \#X: \boxed{kx} + \#Y: \boxed{ky} \qquad (d)$$

$$\&\ \{Z\!\uparrow\boxed{}kz\} \leq^{*} \{X\!\uparrow\boxed{kx}\} \cup \{Y\!\uparrow\boxed{ky}\}\ . \qquad (e)$$

This invariant can be achieved initially by making the processed parts all empty, and it will imply the goal of the program when the unprocessed parts are all empty, which - by (d) - will occur when the unprocessed part of Z is empty. Thus we can use a program of the form:

```
begin integer kx, ky, kz;
kx := lower X; ky := lower Y; kz := lower Z;
while kz ≤ upper Z do "Copy One Element"
end .
```

In "Copy One Element", a single element will be moved from the unprocessed part of X or Y into the processed part of Z. To preserve condition (e) the element to be moved must be the smallest member of $\{X\!\uparrow\boxed{kx}\} \cup \{Y\!\uparrow\boxed{ky}\}$. Since both X and Y are ordered, this will be the smaller of the leftmost unprocessed elements, X(kx) or Y(ky), providing both unprocessed parts are nonempty. However, if only one unprocessed part is nonempty, its leftmost element will be the element to be moved. Thus:

```
"Copy One Element" ≡
    if (if kx > upper X then false else
        if ky > upper Y then true else
        X(kx) ≤ Y(ky))
    then "Copy X" else "Copy Y" ,
```

where, prior to executing "Copy X",

$$IX \equiv Z: \boxed{kz|} \;\&\; X: \boxed{kx|} \qquad (f)$$

$$\&\ X(kx) \leq^{*} \{X\!\uparrow\boxed{kx}\} \cup \{Y\!\uparrow\boxed{ky}\} \qquad (g)$$

will hold as well as the invariant I.

Thus (e) will be preserved if "Copy X" moves X(kx) out of the unprocessed part of X and into the processed part of Z. Moreover, (e) insures that X(kx) will be larger or equal to the elements which have previously been moved into Z. Thus the ordering (b) will be preserved if X(kx) is placed at the right of the processed part of Z. Therefore:

"Copy X" ≡

begin Z(kz) := X(kx); kx := kx + 1; kz := kz + 1 end ,

and by a similar argument

"Copy Y" ≡

begin Z(kz) := Y(ky); ky := ky + 1; kz := kz + 1 end .

Formally, in the notation of Reference 2, "Copy X" must meet the specification

$$I \;\&\; IX\ \{\text{"Copy X"}\}\ I\ .$$

To exemplify the application of the various laws we have stated, we give a formal proof of this specification. The assignment axioms (6.5) and (6.6) imply I' {"Copy X"} I, where

$$I' \equiv I|_{kz \to kz+1}|_{kx \to kx+1}|_{Z \to [Z \mid kz \mid X(kx)]}$$

$$= X: \boxed{kx} \;\&\; Y: \boxed{ky} \;\&\; Z: \boxed{kz} \qquad (a')$$

$$\&\ \mathrm{ord}_{\leq}\,[Z \mid kz \mid X(kx)]\uparrow \boxed{kz} \qquad (b')$$

$$\&\ [Z \mid kz \mid X(kx)]\uparrow \boxed{kz} \sim X\uparrow \boxed{kx} \frown Y\uparrow \boxed{ky} \qquad (c')$$

$$\&\ \#Z: kz\boxed{} = \#X: kx\boxed{} + \#Y: \boxed{ky} \qquad (d')$$

$$\&\ \{[Z \mid kz \mid X(kx)]\uparrow \boxed{kz}\} \leq^{*} \{X\uparrow kx\boxed{}\} \cup \{Y\uparrow \boxed{ky}\} \qquad (e')$$

(Here we have written kx| instead of |kx+1 and kz| instead of |kz+1.) Thus we must show that I & IX implies I', i.e., that lines (a) through (g) imply (a') through (e').

By the rule (2.9) of substitution, (a) and (f) imply

$$X: \boxed{kx} \;\&\; Y: \boxed{ky} \;\&\; Z: \boxed{kz} \qquad (h)$$

which, by the rule (2.7) of erasure, implies (a') as well as various partition diagrams used in the sequel. In particular, by (2.2) and (2.1), $X: \boxed{kx}\boxed{}$ implies $\#X: \boxed{kx} = \#X: kx\boxed{} + 1$, and $Z: \boxed{kz}\boxed{}$ implies $\#Z: \boxed{kz} = \#Z: kz\boxed{} + 1$, so that (d) implies (d').

Next, we have

$$[Z \mid kz \mid X(kx)]\restriction \boxed{kz} \simeq [Z \mid kz \mid X(kx)]\restriction \boxed{}kz \frown [Z \mid kz \mid X(kx)]\restriction \boxed{kz}$$

$$(Z: \boxed{}\boxed{kz}),\ (8.7)$$

$$= Z\restriction \boxed{}kz \frown [Z \mid kz \mid X(kx)]\restriction \boxed{kz} \qquad (6.4)$$

$$\simeq Z\restriction \boxed{}kz \frown X\restriction \boxed{kx} \qquad (7.7),(8.4),(6.2)$$

$$\sim (X\restriction \boxed{}kx \frown Y\restriction \boxed{}ky) \frown X\restriction \boxed{kx} \qquad (c),(8.6)$$

$$\sim (X\restriction \boxed{}kx \frown X\restriction \boxed{kx}) \frown Y\restriction \boxed{}ky \qquad (8.3),(8.5),(8.6),(7.4)$$

$$\simeq X\restriction \boxed{kx} \frown Y\restriction \boxed{}ky \qquad (X: \boxed{}\boxed{kx}),\ (8.7)$$

which establishes (c'), and also

$$[Z \mid kz \mid X(kx)]\restriction \boxed{kz} \simeq Z\restriction \boxed{}kz \frown X\restriction \boxed{kx}\ . \qquad (i)$$

Then

$$\{[Z \mid kz \mid X(kx)]\restriction \boxed{kz}\}$$

$$= \{Z\restriction \boxed{}kz \frown X\restriction \boxed{kx}\} \qquad (i),(7.4),(7.5)$$

$$= \{Z\restriction \boxed{}kz\} \cup \{X\restriction \boxed{kx}\} \qquad (8.8)$$

$$= \{Z\restriction \boxed{}kz\} \cup \{X(kx)\} \qquad (3.8)$$

$$\leq^{*} \{X\restriction \boxed{kx}\} \cup \{Y\restriction \boxed{ky}\} \qquad (e),(g),(4.3a)$$

$$= \{X\restriction \boxed{kx} \frown X\restriction kx\boxed{}\} \cup \{Y\restriction \boxed{ky}\} \qquad (X: \boxed{kx}),\ (8.7)$$

$$= \{X\restriction \boxed{kx}\} \cup \{X\restriction kx\boxed{}\} \cup \{Y\restriction \boxed{ky}\} \qquad (8.8)$$

so that (4.1a) and (4.1b) give (e') and

$$\{Z\restriction \boxed{}kz\} \leq^{*} \{X\restriction \boxed{kx}\}\ . \qquad (j)$$

Finally, (3.1), (2.1), and (4.6) imply $\operatorname{ord}_{\leq} X\restriction \boxed{kx}$, which with (b), (j), and (8.9) implies $\operatorname{ord}_{\leq}(Z\restriction \boxed{}kz \frown X\restriction \boxed{kx})$, which with (i) and (7.8) implies (b').

10. Multidimensional Arrays

Although the concepts we have presented were developed and tested in the context of one-dimensional arrays, most of them extend to the multidimensional case. The major additional concept which is needed is the Cartesian product:

$$S_1 \times \ldots \times S_n \equiv \{\langle i_1, \ldots, i_n \rangle \mid i \in S_1 \;\&\; \ldots \;\&\; i_n \in S_n\} .$$

A Cartesian product of intervals is called a block. The values of the array declared by τ array $X(a_1 : b_1, \ldots, a_n : b_n)$ are functions whose domain is the block $\boxed{a_1 \quad b_1} \times \ldots \times \boxed{a_n \quad b_n}$.

It is evident that the values of subarrays of X such as rows and columns are restrictions of X to certain blocks. For example, the following asserts that $\langle i, j \rangle$ is a saddle point of the two-dimensional array X:

$$\{X \upharpoonright (\boxed{i} \times \boxed{\quad}\,)\} \leq^* X(i, j)$$
$$\;\&\; X(i, j) \leq^* \{X \upharpoonright (\boxed{\quad} \times \boxed{j}\,)\} .$$

11. Conclusion

The contents of this paper is only a small beginning. It is largely limited to one-dimensional integer-subscripted arrays, and even within this domain further study is certain to produce significant extensions and changes. But we have gone far enough to demonstrate the value of the underlying approach: We have formulated concepts, laws, and notations which are powerful tools for the precise yet intelligible description of a significant aspect of programming.

Hopefully, this work suggests guidelines for further progress: One should focus upon particular mechanisms such as arrays rather than generalities which pertain to all computation. Concepts and laws are more fundamental than notation per se, and should reflect intuitive understanding. Most important, the crucial test is the ability to describe real programs in a way which is not only precise but also intelligible to the human reader.

APPENDIX

Proof of Proposition (2.3)

We leave it to the reader to verify that either $a_0 \le a_1 \le \ldots \le a_n$ or $a_0 \ge a_1 \ge \ldots \ge a_n$ implies $a_0 \boxed{\; a_1 \mid \ldots \; a_n}$. The following proof of the converse was found by F. L. Morris.

Suppose $a_0 \boxed{\; a_1 \mid \ldots \; a_n}$. From (2.2) we have

$$\#\, a_0 \boxed{\; a_n} = \sum_{i=1}^{n} \#\, a_{i-1} \boxed{\; a_i} \quad , \tag{a}$$

where

$$\#\, a \boxed{\; b} = \underset{\sim}{\text{if}}\ b - a \ge 0\ \underset{\sim}{\text{then}}\ b - a\ \underset{\sim}{\text{else}}\ 0$$

is always nonnegative and is zero iff $a \boxed{\; b}$ is empty. For arbitrary a_i's simple cancellation gives

$$a_n - a_0 = \sum_{i=1}^{n} a_i - a_{i-1} \; .$$

Then subtraction of (a) from both sides gives

$$f(a_0, a_n) = \sum_{i=1}^{n} f(a_{i-1}, a_i) \quad , \tag{b}$$

where

$$f(a, b) = b - a - \#\, a \boxed{\; b} = \underset{\sim}{\text{if}}\ b - a \ge 0\ \underset{\sim}{\text{then}}\ 0\ \underset{\sim}{\text{else}}\ b - a$$

is always nonpositive and is zero iff $a \boxed{\; b}$ is regular.

The interval $a_0 \boxed{\; a_n}$ must be either empty or regular (or both). Suppose it is empty. Then (a) asserts that a sum of nonnegative terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1} \boxed{\; a_i}$ is empty, and $a_{i-1} \ge a_i$.

On the other hand, suppose $a_0 \boxed{\; a_n}$ is regular. Then (b) asserts that a sum of nonpositive terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1} \boxed{\; a_i}$ is regular, and $a_{i-1} \le a_i$.

ACKNOWLEDGEMENTS

I am indebted to the members of IFIP Working Group 2.3, who have provided motivation, inspiration, and helpful criticism. I am also grateful for the hospitality of the University of Edinburgh and the support of the Science Research Council during the period when this paper was written.

REFERENCES

1. Floyd, R. W. "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics 19, American Mathematical Society, Providence (1967), pp. 19-32.

2. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming," Comm. ACM 12, no. 10, October 1969, pp. 576-581.

3. Hoare, C. A. R. "Proof of a Program: FIND," Comm. ACM 14, no. 1, January 1971, pp. 39-45.

4. Dijkstra, E. W. A Discipline of Programming, Prentice-Hall, 1976.

5. Hoare, C. A. R. "Notes on Data Structuring," in Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press (1972), pp. 83-174.

6. Hoare, C. A. R., and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, 1973, pp. 335-355.

7. Hoare, C. A. R., "A Note on the FOR Statement," BIT 12, no. 3, 1972, pp. 334-341.

8. Cooper, D. C. "Proofs about Programs with One-Dimensional Arrays," unpublished.

9. Burstall, R. M. "Some Techniques for Proving Correctness of Programs which Alter Data Structures," Machine Intelligence 7, November 1972, pp. 23-49.

Section 9

EVALUATION MODELS FOR INDEX SEQUENTIAL FILES

Amrit L. Goel
Yuan Liu

Syracuse University

Abstract

The purpose of this report is to develop and illustrate the use of probabilistic models for times spent in various operations on index sequential and other simpler files. Various criteria for evaluating the performance of a file organization are described and the problem of tackling variability in performance is discussed and a detailed description of an index sequential file and operations on the file is given. Models for read, add, update and delete operations on an index sequential file are developed and an explanation of the use of basic models for special cases is presented. Based on these models, expressions for processing times for sequential and direct files are developed. Methods for numerical transformations of random variables needed for these models are also discussed.

TABLE OF CONTENTS

1. PERFORMANCE OF FILE ORGANIZATIONS

1.1 Criteria for Performance Evaluation.

There are three important criteria commonly used to evaluate a file organization: storage cost, processing cost, and response time. The storage cost is the cost of storing the file on some storage medium; the processing cost is incurred in doing various operations on the file and includes the cost of utilizing CPU, core memory, I/O channel, other I/O control devices, etc.; the response time is the time elapsed from the receipt of a request for a certain operation to the end of that operation.

Basically, the storage cost is fixed and the processing cost is variable. The sum of these two should be as small as possible. On the other hand, the response time is only required to be below a certain value. Generally, more storage can be used to decrease processing cost and vice versa. Also, more storage and/or processing equipment can be used to reduce response time. To choose a proper file organization for a logical file, one tries to minimize the total cost, while keeping the response time below an acceptable value.

Given that a file is stored using a certain file organization, it is relatively easy to estimate the storage cost. The portion of processing cost that is fixed is also relatively easy to estimate. The variable portion of the processing cost is proportional to the equipment cost and the elapsed time in using that equipment. Total processing cost is the sum of processing costs on all the equipments. Usually, the toal elapsed time, or response time, is a good indicator of total processing time.

The response time consists of actual processing time and wait time, if any. The latter is largely dependent on the environment and is very difficult to estimate from the file organization alone. Furthermore, wait time is relatively unimportant in choosing a file organization. Ignoring wait time, the response time represents the total processing time.

Due to its central importance we will concentrate on the determination of processing time in evaluating the performance of a file organization.

1.2 Variability in Performance

A file organization will frequently yield different processing times in different situations. For example, the read time in a sequential file depends on the relative position of the desired record in the file. It takes the longest time if the desired record happens to be the last one in the file.

In general, the variability in the performance of a file organization comes from: (1) variability in the file properties such as the number of records in a file, the specific records in a file (when the number of records is fixed), the key value distribution of potential records, the sizes of records (if not fixed-length), and so on; (2) variability in the environment, for example, the nature of other activities on the same device; (3) variability in the history of operations on a file, for example, if the same records are added to and deleted from a hash file in different sequences, different physical files will result, yielding different performances.

In order to adequately determine the performance of a file organization, such variability must be explicitly accounted for by using measures such as variance and higher moments. However, in most applications only averages are employed. Sometimes, values for best and worst cases are also used. Part of the reason for such practices is that determination of variance and higher moments is not only difficult but is almost impossible in many situations.

The models developed in this report permit determination of explicit measures of variability of processing time for index sequential and some simpler files.

2. INDEX SEQUENTIAL FILES

2.1 An index sequential file consists of a data file and a hierarchy of index files. We will restrict ourselves to certain implementations of the index sequential files so that the derived models will cover common implementations and yet will be specific enough to be useful.

Each level of index file is divided into a collection of index subfiles which are stored as ordered sequential files. We will consider the case where these index subfiles are ordered physical sequential files.

The data file is divided into a number of data subfiles, and an index record at the lowest level points to each of these subfiles. It is not necessary, although desirable, in many cases to maintain the data file such that the data subfiles are ordered by a certain condition, which may or may not coincide with the ordering condition of the index files. A data subfile is stored, then, using any file organization that is appropriate.

We will consider the following implementation of the data file that encompasses all the common index sequential files. Each data subfile is further divided into a "prime data subfile" and an "overflow data subfile" which are both sequential files. An index record at the lowest level points to each prime or overflow data subfile. The collection of all prime data subfiles is called the "prime data file" and the collection of all overflow data subfiles is called the "overflow data file". Note that the implementation where there is no overflow data file is a special case.

Referring to subfiles, the phrase "of the same kind" will be used to mean "of the prime data file, of the overflow data file, or of an index file at a certain level."

2.2 Storage Medium

We will consider files stored on disks, which are commonly used for on-line systems. Such a storage medium is in units of "packs", each having one or more cylinders and a number of tracks on each cylinder. On each pack there is one read/write assembly that is, at any moment of time, located at a particular cylinder, at all the tracks of that cylinder, and at an angular position. The angular position changes at a constant speed which is a characteristic of the disk. However, the cylinder position does not change except possibly to read or write a record. The assembly must move to the correct cylinder and the correct angular position to read or write a record. The packs operate independently of each other.

Note that this description includes the head-per-track disk devices as a special case where the time to mover from one cylinder to another may be considered to be always zero.

Thus, the time to read or write a record at an address is the sum of a seek time (to position to the cylinder), a latency time (to position to the angular position), and a transmission time (to actually read or write the record). The seek time is a function of the cylinder addresses of the last record accessed on the same pack and the desired record. The latency time is a function of the angular position of read/write assembly at the time when it arrives at the correct cylinder and the desired angular position. The track address of the desired record is irrelevant for timing purposes.

3. CONVENTIONS AND NOTATIONS

3.1 Subscripted Variables

The following variables may have subscripts listed later.

A add time

D delete time

K seek time; time to position the read/write assembly of a disk pack to a specific cylinder

L latency; time to position the read/write assembly to the start of a record

M relative position (a number) of a record in a file after operation

M' the number of records shifted in a physical sequential file in an operation

N the number of records in a file. When referring to a physical file, it may or may not include deleted records in it.

R read time; time to read a record

T transmission time; time to transmit a record, including the time to pass a gap between records.

U update time; time to update a record

3.2 Non-subscripted Variables

H search time; time to read and examine a number of records in order to find a record meeting a given condition.

p	a probability
Q	number of levels in an index sequential file
r	a probability

3.3 Subscripts

The following symbols are used as subscripts to denote that the subscripted variables pertain to a file or a certain file organization or a file of a certain kind in an index sequential file.

D	direct file
F	an index file at any level, a prime data file or an overflow data file
i	an index file at level i
Numerals 1,2,etc	index file at the indicated level
P	prime data file
Q	the top level index file
S	a sequential file
V	an overflow data file
X	an index sequential file

3.4 A Special Notation

The notation $\sum_{M_V} Y$ or $\sum_{1}^{M_V} Y$ is used to denote the sum of M_V random

variables which are identically, independently distributed as Y, where M_V may, in general, be any integer and Y is a given random variable. That is, $\sum_{M_V} Y \equiv \sum_{i=1}^{M_V} Z_i$, where each Z_i is distributed as Y and all Z_i's are identically, independently distributed and Z_i's are not explicitly referred to.

4. DEVELOPMENT OF BASIC EXPRESSIONS

In this section we develop basic expressions for read, add, delete and update operations on an index sequential file.

4.1 Read Time

The read operation consists of two steps:

(i) Search in each level of the index files for the address of the desired data subfile; and

(ii) Search in the prime data subfile or the overflow data subfile as indicated by step (i).

The expressions for the times spent in these steps are derived below.

The search for the address starts in the top-level (level Q) index file. The time spent here consists of seek time K_Q, latency time L_Q and transmission time for M_Q records, $M_Q T_Q$. The quantities K_Q, L_Q and M_Q are random variables and hence the time spent at level Q is a random variable given by $(K_Q + L_Q + M_Q \cdot T_Q)$. Note that the individual terms in this sum may or may not be statistically independent.

Next, we search at levels (Q-1), (Q-2), ..., 1. The time spent at level i is given by $(K_i + L_i + M_i T_i)$. Hence, the total time spent in the index files is

$$TTSIF = \sum_{i=1}^{Q} (K_i + L_i + M_i T_i) \qquad (1)$$

With probability p, we search for the record in a prime data subfile, a physical sequential file. The time spent here consists

of seek time K_P, latency time L_P and read time for M_P consecutive records, $M_P T_P$. Thus, the total time spent in the prime data subfile is given by the random variable

$$TTSIPDS = (K_P + L_P + M_P T_P) \tag{2}$$

With probability q, we search for the desired record in an overflow data subfile, which is a pointer sequential file. The time spent here is the sum of seek time K_V and the time to read M_V records in the file, $\sum_{1}^{M_V}(L_V + T_V)$. The latter item is the sum of M_V identically independently distributed random variables. The total time is thus,

$$TT = K_V + \sum_{1}^{M_V} (L_V + T_V). \tag{3}$$

When we combine the above expressions, the read time for a single record is given by the random variable

$$R = \sum_{i=1}^{Q} (K_i + L_i + M_i T_i) + \begin{cases} p: & K_P + L_P + M_P T_P \\ q: & K_V + \sum_{1}^{M_V}(L_V + T_V) \end{cases} \tag{4}$$

where the second term is a discrete mixture of two random variables. A definition of discrete mixtures is given in Appendix I.

4.2 Add Time

The add operation consists of two steps:

(i) Search the index files to determine the address of the data subfile where the given record should be added, and

(ii) Add the given record to the data subfile as indicated by step (i).

The expressions for the times spent in these steps are derived below.

(a) The time spent in step (i) is the same as for step (i) in the read operation and is given by Equation (1).

(b) With probability p, the given record is added to a prime data subfile. Let the added record, after the add operation, be the M_pth record of the subfile.

In order to maintain ordering, a number, M_p', of records must be moved to make space for the given record. Note that M_p' is a random variable taking values, 0, 1, 2, ..., N_p, where N_p is the number of records in a prime data subfile and may also be a random variable. Note that the time needed to replace one record in a sequence of consecutive records is $1 + T_p$. If there is space in the prime data subfile to accommodate the given record, and only one record in addition to the given record can be held in core, the time taken to move M_p' records is $(1 + T_p)M_p'$. However, if the size of the prime data subfile is limited and the added record will cause the limit to be exceeded, one or more records may have to be moved to other, perhaps newly created, prime or overflow data subfiles. The time required for this case will be governed by the specific technique used to handle such situations. Here, we consider only the case where there is enough room in the prime data subfile to accommodate the new record.

The total time spent in the prime data subfile is, thus, the sum of seek time K_p, latency time to reach the first record L_p,

search time to read M_p records $M_p \cdot T_p$, and time to move M'_p records $M'_p \cdot (1 + T_p)$, and is given by

$$TTSPDS = (K_p + L_p + M_p \cdot T_p + M'_p(1 + T_p). \qquad (5)$$

(c) With probability q, the given record is added to an overflow data subfile, which is a pointer sequential file, where adding a given record between two consecutive records A and B (B follows A) requires A to be updated and the given record written.

The time needed to add a record to an overflow data subfile consists of seek time, K_V, search time for $M_V - 1$ records, $\sum_{1}^{M_V-1}(L_V + T_V)$, time to read record M_V and update record $M_V - 1$, 1, time to write the added record, $L_V + T_V$.

The total time spent here is thus given by

$$TTAODS = K_V + \sum_{1}^{M_V-1} (L_V + T_V) + 1 + (L_V + T_V) \qquad (6)$$

Note that if there is a backward pointer, extra time will be needed for updating the next record and should be included in the above time.

Combining the expressions from (a), (b) and (c), the time for adding a single record under the conditions described above is given by the random variable

$$A = \sum_{i=1}^{Q} (K_i + L_i + M_i T_i) + \begin{cases} p: & K_p + L_p + M_p T_p + M'_p(1 + T_p) \\ q: & K_V + \sum_{1}^{M_V} (L_V + T_V) + 1. \end{cases} \qquad (7)$$

where the second expression is a discrete mixture of two random variables (see Appendix I).

4.3 Update Time

The update operation consists of reading the record to be updated followed by writing over it the updated record. The total time is given by the random variable U,

$$U = R + 1 \tag{8}$$

where R is the read time given in Section 4.1.

4.4 Delete Time

The delete operation consists of reading the record to be deleted and writing over it a deleted record. This requires one read operation and one disk revolution. Hence, the delete time is given by a random variable

$$D = R + 1 \tag{9}$$

where R is the read time given in Section 4.1.

An alternative to this delete operation is to remove the record from the subfile, thus decreasing the number of records in the subfile by 1. The extra time needed to remove the record can be obtained by using an approach similar to the one used for obtaining read and add times in Section 4.1 and 4.2, respectively. This time is not reflected in the above expression.

5. NOTES ON THE APPLICATION OF THE BASIC MODELS - SPECIAL CASES

The purpose of this section is two-fold: (1) to derive timing expressions for some useful special cases, and (2) to point out how the basic expressions for the index sequential file are modified for specific purposes.

Each of these cases specifies one aspect of the general index sequential file. If two or more aspects are specified at a time, or some other aspect is specified, this section should be helpful in showing the general approach to be used for the given situation.

5.1 Index Sequential File on a Single Pack

If an index sequential file is stored on a single pack, then in order to carry out a series of operations on random, individual records, the read/write assembly moves only from the top level index file to index files at levels (Q-1), (Q-2), ..., 1 and then to either an overflow or a prime data subfile and back to the top-level index file. In this case the seek time to each "kind" (as defined in Section 3.3) of file, say to an index file at level 1, is more conveniently denoted by $K_{2,1} \equiv K_1$, indicating the seek time from a record in an index subfile at level 2 to the beginning of an index subfile at level 1. The quantities $K_{i,i-1}$, $K_{1,P}$, $K_{1,V}$, $K_{V,Q}$, $K_{P,Q}$, etc. are similarly defined as the seek times for a record in a subfile of a certain kind to the beginning of a subfile of another kind.

The seek time to the top-level index file K_Q needs some clarification. Since the random variable is dependent on whether the last record is in the

prime data file or the overflow data file, it is best separated into two terms $K_{P,Q}$ and $K_{V,Q}$ and combined, respectively, with the times spent in the prime and the overflow data files.

Using arguments similar to those for expression (4) in the basic model, the read time for an index sequential file residing on a single pack is given by the random variable

$$L_Q + M_Q \cdot T_Q + \sum_{i=1}^{Q-1} (K_{i+1,i} + L_i + M_i \cdot T_i) + \begin{cases} p: K_{1,P} + L_P + M_P \cdot T_P + K_{P,Q} \\ q: K_{1,V} + \sum_{1}^{M_V} (L_V + T_V) + K_{V,Q} \end{cases} \tag{10}$$

Similarly, the add time for an index sequential file residing on a single pack is given by the random variable

$$L_Q + M_Q T_Q + \sum_{i=1}^{Q-1} (K_{i-1,i} + L_i + M_i \cdot T_i) + \begin{cases} p: K_{1,P} + L_P + M_P \cdot T_P + M_P' \cdot (1 + T_P) + K_{P,Q} \\ q: K_V + \sum_{1}^{M_V+1} (L_V + T_V) + 1 + K_{V,Q} \end{cases} \tag{11}$$

The relationships between update and delete times and read time are the same as in the basic models.

5.2 Number of Records Searched in a Subfile

The number of records that must be read before the desired record is found in a subfile is, in general, a random number.

Consider a subfile of kind X, where X = (an integer between 1 and Q), P, or V, indicating an index subfile at a certain level, a prime data subfile, or an overflow data subfile, respectively. If the desired record exists in the subfile, and if all records in a subfile of kind X have equal probability of being used for read, add, delete or update operations, then the number of records searched, M_X, has a discrete uniform distribution over the range $(1,N_X)$, where N_X is the number of records in this subfile. On the other hand, if the desired record does not exist in the subfile, all the records will have to be searched before the operation is over, and hence $M_X = N_X$.

Let r_X be the probability that the desired record exists in subfile X. Then M_X is given by the following discrete mixture:

$$M_X = \begin{cases} r_X : i, \; i = 1,2, \ldots, N_X \\ 1 - r_X : N_X \end{cases} \tag{12}$$

5.3 Interference from Other Files

If there are other files that share the same pack, it is possible that the pack has to leave a cylinder of the index sequential file under consideration and return to it later. Effectively, the seek time is increased as described below.

Let F denote all other files that share the same device with a file of kind X, where X = (an integer between 1 and Q), P or V. The random variable K_X is the sum of (i) seek time from a file of kind X to a file of F, (ii) the time spent in a file of F, and (iii) the seek time from F to X. The distributions of the above three variables may be estimated by noting the distributions of the appropriate cylinder addresses and the overall processing time in a file of F.

5.4 Mixed Operations

If a series of different operations are carried out on the index sequential files, the time to process a single, random record is given by the discrete mixture (p_R:R, p_A:A, p_U:U, p_D:D), where p_R, p_A, p_U, and p_D are the probabilities of read, add, update and delete operations, respectively. Each operation may have its own distributions of the numbers of records searched, cylinder addresses (for seek times), etc. This is easily accounted for by considering the discrete mixtures of the appropriate seek times in different situations.

6. TIMING EXPRESSIONS FOR SEQUENTIAL FILES

Since an index sequential file is composed of a number of sequential files, the same approach used before in deriving the timing expressions for index sequential files may be applied to sequential files to obtain timing expressions.

The timing expressions for read, add, delete, and update operations on sequential files are given in this section, along with some notes on applying these expressions to actual files.

6.1 Read Time

The time to read one record from a sequential file is equal to the sum of (i) seek time K and (ii) search time H to read and examine M records until the desired record is read or until every record is read. Thus, the read time R = K + H.

The seek time K is determined by the cylinder address (in general, a random variable) of the read/write assembly before the read operation and the cylinder address of the first record in the file.

The search time, H, is different for physical sequential and pointer sequential files. For the former, search time is given by H = L + MT, where L is latency to reach the first record and T is the transmission time of a record and may also be a random variable if the record length is not fixed. For a pointer sequential file, search time is given by $H = \sum_{M}(L + T)$, where L is the latency to reach the first read or the next record indicated by the pointer field and T is the transmission

time as before.

The latency, L, is a function of the position of the read/write assembly relative to a desired record and may usually be approximated by a uniform random variable in the interval (0,1), provided the timing unit is the revolution time of the disk.

Note that M is determined by (i) the probabilities of reading the records in the sequential file, (ii) the probability of the desired record being in the file, and (iii) whether the records are ordered by a certain condition.

6.2 Add Time

The add time is the sum of (i) time to locate the address where the new record is stored, (ii) the time to store the record at the found address, and (iii) the time to adjust the file to maintain the integrity of the file.

There are many variations of the add operation. A common case is considered here as an example. It is assumed that (i) the records in the file are ordered by a key, (ii) the records are deleted from the file by modifying them to a special record, (iii) when the limit to the number of records that can be stored in the file is exceeded, a new sequential file is created elsewhere and the new record is stored there, and (iv) only one record in addition to the new record may be held in core.

Let K and L be the seek and latency times, respectively, to prepare for reading the first record, N be the number of records (excluding deleted records) in the file before addition, M be the record number of the new

record (including deleted records) and M' be one less the number of records between the Mth record before addition and the first deleted record after it (inclusive). Thus, if the new record must be added between records 2 and 3 (including deleted records) to satisfy the ordering condition, and record 3 is not a deleted record but record 4 is, then M = 3 and M' = 1.

The total time, shown in the breakdown given above, is (i) $K + L + \sum_{M} T$ (ii) 1, and (iii) $\sum_{M'} T + \begin{matrix} p: C \\ 1-p: 0 \end{matrix}$, where p is the probability that the capacity is exceeded and C is the time to create a new file containing the last record from the original file. Thus

$$A = \sum_{M} T + 1 + \sum_{M'} T + \begin{cases} p: C \\ 1-p: 0 \end{cases} \tag{13}$$

6.3 Update Time

The update time is given by

$$U = R + 1. \tag{14}$$

6.4 Delete Time

The delete time varies with the delete operation. For the case where a record to be deleted is simply marked so, the delete time is equal to update time and, hence,

$$D = R + 1. \tag{15}$$

7. TIMING EXPRESSIONS FOR DIRECT FILES

The read time, R, is the sum of (i) seek time K and latency time L before reaching the desired record and (ii) transmission time T.

The seek time K is a function of the number of cylinders crossed, which is the absolute difference between the cylinder Address C (a random variable) of the read/write head at the beginning of the add operation and the cylinder address C' (another random variable) of the desired record. Let s(x), a characteristic of the disk device, be the seek time to cross x cylinders. Then the read time is

$$R = s(|C - C'|) + L + T. \tag{16}$$

The add time is the sum of (i) the read time for the existing record at the given address, and (ii) the time to write the given record at the given address if allowed. Thus,

$$A = R + 1 \tag{17}$$

The update time is U = R + 1, and the delete time is D = R + 1, since a record to be deleted is simply updated.

8. METHODS FOR NUMERICAL TRANSFORMATIONS OF RANDOM VARIABLES

In the models derived in the previous sections various transformations of random variables are involved, such as the summation of a number of random variables (some discrete, some continuous), the discrete mixture, and the summation of N random variables where N is also a random variable. There are no existing methods which are simple and flexible enough to make the application of these models practical.

The purpose of this section is to discuss various computational methods possible for this purpose and to find a computational strategy that (i) minimizes the computational efforts, (ii) can be used for cases where complete or partial data about the component random variables is available, and (iii) can work with totally arbitrary distributions that may be discrete, continuous, or mixed.

8.1 Approximating Density Functions

In many cases the values of the density function of each random variable may be approximated at a certain number of abscissa points. The disadvantages of this approach are (i) the density function of each random variable must be estimated in great detail, and (ii) a large amount of computation and storage is required in many situations (e.g. a very irregular discrete density function.)

Sometimes flexible distributions, such as beta distribution, may be used to fit a density function. This method is limited to rather well behaved distributions and cannot be applied to totally arbitrary density

functions. Furthermore, transformations other than summation are difficult and may introduce different (e.g., non-beta) distributions. Therefore, they are not suitable for the type of situation encountered in the present context.

8.2 Histogram Method

Another possible method is to use a step function, called "histogram function", to approximate a density function. Given a set of points a(1), a(2), ..., a(n), the histogram function is a step function, which is chosen such that the integration between a(i) and a(i+1) of the histogram function is equal to that of the approximated density function.

A procedure to obtain an approximate histogram function for the sum of two random variables, whose individual histogram functions are given, is described in Appendix II.

Various difficulties and inaccuracies are introduced by the use of this method. First, not every density function can be represented exactly by a histogram function. Secondly, the convolution of two histogram functions cannot be represented exactly by a histogram function. (It is a piecewise linear function.)

In order to test the feasibility of this method, a number of standard normal random variables were added (see Appendix II). The errors introduced by the method seem to increase linearly with the accumulative number of additions performed in a histogram. This is generally satisfactory. However, if the interval size is kept constant, the number of intervals increases multiplicatively with each addition performed, even if intervals with small weights are discarded. If the interval size is increased after each addition, more complicated procedures must be used and greater error will be introduced. A detailed discussion of this aspect is given in Appendix II.

The histogram method is applicable for discrete distributions which are irregular, but requires a large amount of storage in many cases. Although the histogram method has the advantage of being applicable for cases where only limited information is available, it was abandoned in this study because the method becomes much more complicated for transformations other than addition and the computation efforts can become too costly.

Various other methods of approximating a function exist but are not useful because we must be able to handle totally arbitrary functions that may be continuous, discrete, or mixed.

8.3 Moments Method

It is well known that a random variable is completely defined if all of its moments, if existing, are known. Also, if only the first few moments are known, partial information about the shape of the distribution may be deduced. If the transformations may be conducted solely on the first few moments, the method may be used to gain information about a random variable depending on the information available. If more moments are needed, one simply tries to obtain more information (in terms of more moments) about the component random variables and applies the method again.

Formulas for conducting various random variable transformations on the moments are presented in Appendix III. The transformations include summation, multiplication, and discrete mixture, which are needed for the index sequential file model derived previously.

Once the moments of a random variable are obtained, various statistics such as the mean, variance, skewness, kurtosis, etc. may be estimated readily. In addition, the approximate density function may be estimated from the first four moments and estimations of the density function can be obtained (see, for example, [Royden 1953] and [Elderton 1969]).

The moments method has the advantages that the computation procedure is simple and the method is flexible in that the number of moments used can be varied according to available data and desired accuracy.

The moments method of carrying out random variable transformations is most appropriate for the index sequential file models derived in the previous sections. This method will be used in several numerical examples to illustrate the use of the models.

REFERENCES

ELDERTON 1969

Elderton and Johnson, "Systems of frequency curves," Chapter 4, "Pearson's system of frequency curves," Cambridge Press, 1969.

ROYDEN 1953

Royden, H.L. "Bounds on a distribution function when its first n moments are given," Annals of Mathematical Statistics, 1953, pp. 361-376.

Appendix I

DISCRETE MIXTURES

1. Definition

The "discrete mixture" Y of n statistically independent random variables $X_1, X_2, \ldots X_n$, with weights of $p_1, p_2, \ldots, p_n$, $\sum_{i=1}^{n} p_i = 1$ is a random variable whose density function f_Y is given by the weighted sum of the density functions f_{X_i} of the "components" $X_1, X_2, \ldots, X_n$. In other words, $f_Y(X) = \sum_{i=1}^{n} p_i f_{X_i}(X)$. The following are notations for a discrete mixture:

$$Y = (p_i:X_i; \quad i = 1, \ldots, n)$$

$$Y = (p_1:X_1, p_2:X_2, \ldots, p_n:X_n)$$

This definition also can be easily modified for the discrete random variables $X_1, X_2, \ldots, X_n$.

2. Illustration

Informally, Y is a random variable which assumes the behavior of X_i with probability p_i, for i = 1,2, ...,n. Note that it is different from the weighted sum of X_i's, namely, $Z = \sum_{i=1}^{n} p_i X_i$, whose density function is the convolution of density functions of $p_i X_i$, i = 1,2,...,n.

Let $X_1 = 1$ with probability 0.5 and $X_1 = 2$ with probability 0.5 and $X_2 = 10$ with probability 0.5 and $X_2 = 20$ with probability 0.5. Note that X_1 and X_2 are discrete mixtures of degenerate random variables.

Let Y be a discrete mixture of X_1 and X_2 with weights 0.2 and 0.8, i.e., $Y = (0.2;X_1, 0.8:X_2)$ and let $Z = 0.2(X_1) + 0.8(X_2)$; then Y = (0.1: 1, 0.1: 2, 0.4:10, 0.4: 20) and Z = (0.25: 8.2, 0.25:8.4, 0.25: 16.2, 0.25:16.4). Clearly, they are different random raviables. The means of

Y and Z are equal, but highre moments are not.

The following theorems can be easily proven by directly using the definitions.

3. Theorem A-1

The moments of a discrete mixture are the weighted sums of the corresponding moments of its components, i.e.

$$\mu_Y(i) = \sum_{j=1}^{n} p_i \, \mu_{X_j}(i)$$

where

$Y = (p_j: X_j;\ j = 1,2, \ldots,n)$ and $\mu_Y(i)$ and $\mu_{X_j}(i)$ are the ith moments about the origin of Y and X_j respectively.

4. Theorem A-2

$Z + (p_i:X_i;\ i = 1,2, \ldots,n)$ is identical to $(p_i: Z + X_i;\ i = 1,2,\ldots n)$ if Z and X_i, $i = 1,2,\ldots,n$ are all statistically independent.

5. Theorem A-3

$(p_1:X, p_2: (q_1: Y, q_2: Z)) = (p_1: X, p_2*q_1: Y, p_2*q_2: Z)$ where X, Y and Z are statistically independent.

Appendix II

USING STEP FUNCTIONS FOR CONVOLUTION

1. Introduction

In order to carry out the transformations indicated in the evaluation model of index sequential files, simple and convenient ways have to be chosen. In this appendix, we will investigate the method of using step functions, called "histogram functions", to approximate density functions by studying the sum of independent random variables.

A "histogram function" hA(i) of a random variable A gives the probability that A lies in interval (a[i], a[i+1]) for i = 1,2,...,m, where a[1], a[2],..., a[m+1] are parameters of the histogram function and constitute the "node set". A specific value hA(i) for a given i is called the "weight" of the histogram function in the interval (a[i], a[i+1]), or at node a[i]. A random variable whose density function is a step function is called a "histogram variable".

We assume the nodes in the node set are all unequal and are numbered in increasing order, and $hA[1]hA[m+1] \neq 0$, without loss of generality.

We further assume that the nodes are all finite. If either or both of the end nodes, a[1] and a[m+1], are infinite, a histogram function can still be considered an approximation to the true density function,

but it causes problems in transformations.

Throughout this appendix, the term "histogram" will be used as an abbreviation for "histogram function". Being an approximation to a density function, a histogram contains only partial information about a random variable. Therefore, it is impossible to obtain the convolution of two random variables given only their histograms.

The convolution of two histograms is not a histogram. However, after choosing a node set, one can always obtain a histogram as an approximation to the convolution of the two density functions.

Such an approximate convolution contains two sources of errors, the first being approximating the original random variables with histograms and the second being approximating the convolution of histograms with a histogram.

The following example shows some of the problems involved in trying to find out the distribution of the sum of two random variables given only their histograms. Later, we will study the errors on a test case using normal variables.

2. An Example

Suppose the following histograms are given for two statistically independent random variables A and B. The true density functions are irrelevant for this example.

A:	node a[i]	-2	-1	0	1	2	3
	hA(i)	.1	.2	.4	.2	.1	

B:	node b[i]	2	4	6	8	10
	hB(i)	.4	.3	.2	.1	

The following table gives information about the joint distribution of A and B and hence about the distribution of C = A + B. Each entry shows three items, p(i,j), s(i,j) and u(i,j), where p(i,j) is the probability that A is in (a[i], a[i+1]) and B is in (b[j], b[j+1]) and s(i,j) and u(i,j) are the numbers enclosed in parentheses representing the lower and upper bounds of corresponding values of C. Each p(i,j) is also the conditional probability that C is in (a[i]+b[j], a[i+1]+b[j+1]), given that A and B are in the indicated intervals, respectively. The headings indicate a[i] and b[j].

Note that the invervals of C, (s(i+j), u(i+j)) overlap.

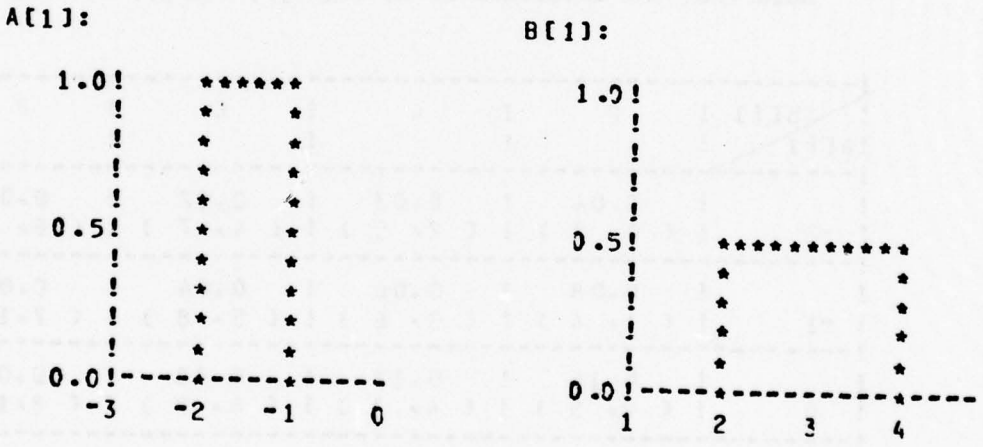
a[i] \ b[j]	2	4	6	8
-2	0.04 (0, 3)	0.03 (2, 5)	0.02 (4, 7)	0.01 (6, 9)
-1	0.08 (1, 4)	0.06 (3, 6)	0.04 (5, 8)	0.02 (7,10)
0	0.16 (2, 5)	0.12 (4, 7)	0.08 (6, 9)	0.04 (8,11)
1	0.08 (3, 6)	0.06 (5, 8)	0.04 (7,10)	0.02 (9,12)
2	0.04 (4, 7)	0.03 (6, 9)	0.02 (8,11)	0.01 (10,13)

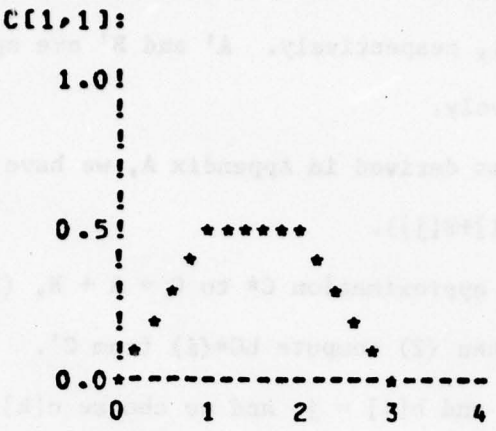
TABLE C-1 p(i,j) and (s(i,j), u(i,j))

Since only partial information is given for the distributions of A and B, it is possible only to obtain an approximate distribution for C. However, if A' and B' represent the random variables defined by the histograms given above, then the sum C' = A' + B' can be obtained exactly. We will try to obtain a histogram from C' as an approximation to the density function of C.

Let A[i] be a uniformly distributed random variable over the interval [a[i], a[i+1]) and B(i) be a uniformly distributed random variable over the interval (b[i], b[i+1]). Also, let C[i,j] = A[i] + B[j].

Consider the sum of A[1] and B[1], which are over intervals (-2,-1) and (2,4), respectively. Let C[1,1] = A[1] + B[1]. Graphically, the density functions A[1], B[1], and C[1,1] are shown as follows:





In Table C-1, C[1,1] contributes (0.04)(0.25) = 0.01 to interval (0,1) for C', (0.04)0.5 = 0.02 to (1,2) and (0.04)(0.25)=0.01 to (2,3).

The weights of each C[i,j] can be similarly broken down. If we sum up all the weights for each interval, the following histogram C':

node	0	1	2	3	4	5	6
hC'(i)	.01	.04	.0975	.15	.1625	.15	.125

node	7	8	9	10	11	12
hC'(i)	.10	.075	.05	.0275	.01	.0025

3. A General Procedure for a Simple Case

Let hA(i) on a[i], i = 0, 1,..., n, and hB(j) on b[j], j = 0, 1,...,m be the two histograms for random variables A and B, respectively. Then, using the notations for discrete mixture in Appendix A, we define A' = (hA(i): A[i], i = 0,1, ...,n), and B' = (hB(j): B[j], j =0,1,2,...,m)

where A[i] and B[j] are uniformly distributed over (a[i], a[i+1]) and (b[j], b[j+1]), respectively. A' and B' are approximations to A and B, respectively.

Using theorems derived in Appendix A, we have C' = A' + B' = (hA(i)hB(j) : A[i]+B[j]).

To obtain an approximation C* to C = A + B, (1) choose a node set for C', and then (2) compute hC*(i) from C'.

If a[i] = i, and b[j] = j, and we choose c[k] = k, k = o, 1, ..., (m+n), then C' = (hA(i)hB(j) : C[i,j]), where C[i,j] = A[i] + B[j] and Pr(C[i,j] is in (i+j, i+j+1) ! A[i] is in (i, i+1) and B[j] is in (j, j+1)) = 0.5hA(i)hB(j), and Pr(C[i,j] is in (i+j+1, i+j+2) ! A[i] is in (i, i+1) and B[j] is in (j, j+1)) = 0.5hA(i)hB(j). Therefore,

$$hC^*(N) = \sum_{i}^{i+j=k} \sum_{j} 0.5hA(i)hB(j) + \sum_{i}^{i+j=k-1} \sum_{j} 0.5hA(i)hB(j)$$

4. A Test Case

A number of normal random variables are added using the approximate convolution procedure described above and the errors involved are studied.

Let N1 be distributed as $N(0,1)$, i.e., normally distributed with mean zero and variance 1, and $Ni = \sum_{j=1}^{i} N1$. Note that Ni is distributed as N(0,i).

Let Ni' be the histogram obtained from Ni using a certain node set.

Let N1* = N1',N2* = N1* +' N1*, N3* = N1* +' N2*, N4* = N1* +' N3*,N8* = N4* +' N4* and N16* = N8* +' N8*, where +' represents the approximate convolution procedure described before. Note that, effectively, Ni*

is obtained by applying the procedure i times to N1*.

We will try to measure the error involved in making simple probability statements, i.e., those in the form of Pr(x is in I) = a, where I is an interval or a union of intervals.

Define

$$ERRi(Nj') = \underset{I \subseteq I[i]}{MAX} \int_{x \in I} |f_{Nj}(x) - f_{Nj'}(x)| dx,$$

where I[i] is the interval starting from the i-th node of the histogram Nj ending at the (i+1)-th node, and f_{Nj} is the density function of Nj, and $f_{Nj'}$ is the density function defined by the histogram Nj'.

Define $ERR(Nj') = \sum_i ERRi(Nj')$ so that ERR(Nj') is an upper bound of error involved in making any simple probability statement using $f_{Nj'}$ as an approximate to f_{Nj}.

Similarly, define

$$ERRi(Nj^*) = \underset{I \subseteq I[i]}{MAX} \int_{x \in I} |f_{Nj}(x) - f_{Nj'}(x)| dx,$$

and $ERR(Nj^*) = \sum_i ERRi(Nj^*)$.

Since Nj* is an approximation to Nj', ERR(Nj') is an upper bound to ERR(Nj*). Err(Nj*) - ERR(Nj') represents the error introduced by the approximate convolution procedure.

The following table shows the numerical results for the case where the interval size is kept constant.

i	ERR(Ni*) x 1000	ERR(Ni') x 1000	(ERR(Ni*)-ERR(Ni')) x 1000
1	9.97	9.97	0
2	7.24	7.05	0.19
3	6.26	5.76	0.50
4	5.53	4.98	0.55
4#	5.54	4.98	0.55
8	4.24	3.52	0.72
16	3.69	2.49	1.47

#This N4* was obtained from N2* +' N2*

TABLE C-2 Calculated errors for a test case

It can be seen that the error introduced by the approximate convolution procedure increases linearly with the effective number of times the procedure is applied. The slope is about 0.00009. This is practical.

If the interval size is increased in the procedure (in order to reduce the number of intervals), then larger errors will be introduced. By keeping the interval size constant, the number of intervals increases linearly with i, even after eliminating intervals with very small weights. This is due to flatter density functions.

5. Discussion

The procedure using histograms is feasible and flexible for adding random variables. In practice, it may be necessary to control either the interval size or the number of intervals of the

approximate histograms. The practicality of non-equidistant node sets needs to be investigated.

For other transformations such as multiplication, the general approach can be extended, but it is probably not as simple and straightforward.

Appendix III

THE MOMENTS METHOD FOR RANDOM VARIABLE TRANSFORMATIONS

1. Notations:

The following notations will be used throughout this appendix.

X[i], i = 1,2,...,n:	independent random variables
Y[i], i = 1,2,...,n:	independent identically-distributed random variables
N:	a discrete random variable (independent of all X[i] and Y[i], i = 1,2,...,n.)
p[i], i = 1,2,... :	p[i] is the probability that N = i.
$\mu_j(i) \equiv \mu_{X(j)}(i)$ or $\mu_{Y(j)}(i)$:	the i-th moment of X[j] or Y[j], depending on the context, with $\mu_0[j]$ being defined to be 1 for all j.
C(i,j), i _ j	equals (i!)/(j!)((i-j)!).
E(X)	the expectation of a random variable X.

2. Sum of Two Random Variables

The i-th moment of Ys = X[1] + X[2] is $\mu s_i = E(Ys^i) = E((X[1] + X[2])^i) = \sum_{k=0}^{i} C(i,k)E(X[1]^k)E(X[2]^{i-k})$, which may be rewritten as

$$\mu s_i = \sum_{k=0}^{i} C(i,k)\mu_k[1]*\mu_{i-k}[2].$$

In particular,

$$\mu s_1 = \mu_1[1] + \mu_1[2],$$

$$\mu s_2 = \mu_2[1] + 2\mu_1[1]\mu_1[2] + \mu_2[2],$$

$$\mu s_3 = \mu_3[1] + 3\mu_1[1]\mu_2[2] + 3\mu_2[1]\mu_1[2] + \mu_3[2], \text{ and}$$

$$\mu s_4 = \mu_4[1] + 4\mu_1[1]\mu_3[2] + 6\mu_2[1]\mu_2[2] + 4\mu_3[1]\mu_1[2] + \mu_4[2].$$

3. Product of Two Random Variables

The i-th moment of $Yp = X[1]X[2]$ is $\mu p_i = E(Yp^i) = E((X[1]X[2])^i) = E(X[1]^i)E(X[2]^i)$. Therefore, $\mu p_i = \mu_i[1] \cdot \mu_i[2]$.

4. Discrete Mixture of Random Variables

The i-th moment of $Yc = (p[j]: X[j] ; j = 1,\ldots,n)$ is $\mu c_i = E(Yc^i)$. Since $Yc^i = (p[j]: X[j]^i; j = 1,\ldots,n)$, $\mu c_i = \sum_{j=1}^{n} p[j]\, E(X[j]^i)$. Therefore $\mu c_i = \sum_{j=1}^{n} p[j]\mu_i[j]$. Note that the notation for a discrete mixture follows that given in Appendix I.

5. Sum of N Identical Random Variables

To derive the i-th moment of $Yr = \sum_{j=1}^{n} Yj$, we first recognize that $Yr = (p[k] : \sum_{j=1}^{k} Yj; k = 1,2,\ldots)$. Therefore, the i-th moment of Yr,

$$\mu r_i = \sum_{k=1}^{\infty} p[k] E\{(\sum_{j=1}^{k} Y_i)^i\}$$

Let μ_{N_i} and μ_{Y_i} be the i-th moment of N and Y_i, for any j, respectively. The following are the formulas for the first four moment of Yr.

$$\mu r_1 = \mu N_1 \, \mu Y_1,$$

$$\mu r_2 = \mu N_1 \, \mu Y_2 + \mu N_2 + \mu Y_1^2 - \mu N_1 \, \mu Y_1^2,$$

$$\mu r_3 = (\mu N_3 - 3\mu N_2 + 2\mu N_2)\mu Y^3 + 3(\mu N_2 - \mu N_1)\mu Y_1 \, \mu Y_2 + \mu N_1 \, \mu Y_3,$$

$$\mu r_4 = (\mu N_4 - 6\mu N_3 + 11\mu N_2 - 6\mu N_1)\mu Y_1^4 + 6(\mu N_3 - 3\mu N_2 + 2\mu N_1)\mu Y_1^2 \, \mu Y_2$$
$$+ 4(\mu N_2 - \mu N_1)\mu Y_1 \, \mu Y_3 + 3(\mu N_2 - \mu N_1)\mu Y_2^2 + \mu N_1 \, \mu Y^4$$

6. Continuous Uniform Random Variable

The i-th moment of the continuous random variable uniformly distributed over the interval (0,1), denoted by U(0,1), is given by 1/(i+1). Specifically $\mu U_1 = \frac{1}{2}$, $\mu U_2 = \frac{1}{3}$, $\mu U_3 = \frac{1}{4}$ and $\mu U_4 = \frac{1}{5}$

7. Discrete Uniform Random Variable

Let Ud(1+n) be a discrete random variable distributed uniformly over 1,2, ..., n, then its first four moments are:

$$\mu Ud_1 = \sum_{i=1}^{n} (1/n)\, i = (n+1)/2,$$

$$\mu Ud_2 = \sum_{i=1}^{n} (1/n)\, i^2 = (n+1)(2n+1)/6$$

$$\mu Ud_3 = \sum_{i=1}^{n} (1/n)\, i^3 = n(n+1)(n+1)/4, \text{ and}$$

$$\mu Ud_4 = \sum_{i=1}^{n} (1/n)\, i^4 = (6n^4 + 15n^3 + 10n^2 - 1)/30.$$

8. Discrete Uniform with Random Upper Bound

Let Ud'(1,N) be a discrete random variable distributed uniformly over values 1,2,...,N, where N is a random variable or, more precisely, Ud'(1,N) ~ (p[i]: Ud(1,i), i = 1,2,...,), where p[i] = Pr(N=i).

The first four moments of Ud'(1,N) are:

$$\mu Ud'_1 = \sum_n p[n](n+1)/2 = 0.5(\mu N_1 + 1),$$

$$\mu Ud'_2 = \sum_n p[n](n+1)(2n+1)/6 = (2\mu N_2 + 3\mu N_1 + 1)/6,$$

$$\mu Ud'_3 = \sum_n p[n](n/4)(n+1)^2 = 0.25(\mu N_3 + 2\mu N_2 + \mu N_1), \text{ and}$$

$$\mu Ud'_4 = \sum_n p[n](6n^4 + 15n^3 + 10n^2 - 1)/30 =$$

$$(6\mu N_4 + 15\mu N_3 + 10\mu N_2 - 1)/30$$

Section 10

FILE ORGANIZATION CONCEPTS

Yuan Liu
Amrit L. Goel

Syracuse University

Abstract

In this report we develop a rigorous framework for describing various file organizations and operations on files. Various terms pertinent to the description of file organizations are defined and a description of sequential, index sequential, and hash files is presented. Various operations on these files are also discussed.

TABLE OF CONTENTS

FILE ORGANIZATION CONCEPTS

In this report we first rigorously define some terms pertinent to the description of file organizations. Then we present a brief description of sequential, index sequential and hash files and describe various operations on these files.

1. BASIC CONCEPTS

A file is a collection of similar records; a record is a collection of field values; and a field value is a unit of data under consideration. These and other pertinent terms are rigorously defined here using set-theoretic concepts. The following notations are used for this purpose.

$\times$ = cartesian product,

$\otimes$ = an abbreviation for a set operation used to define a variable format record, to be defined later,

$\cup$ = union,

$\langle e \rangle$ = empty set,

{ } = set definition by enumeration, and

( ) = ordered tuple or precedence in set definition.

1.1 Fields and Values

Consider a number of sets; these sets are called fields. All other concepts are derived from fields using two set operations, which are union and cartesian product. A field value or value

is defined as a member of a field. A field value is a basic, indivisable unit of data under consideration. For a given problem, a field in one formulation may be considered as an entity defined in terms of different fields in another formulation, so that the field value in the former formulation appears to be divisible.

1.2 Formats

The terms "format," "fixed format," and "variable format" are defined recursively as follows. A format is either a fixed format or a variable format. A fixed format is a set which is either a field or a cartesian product of a finite number of other formats. A variable format is defined in terms of a field, C, the control field, a set of formats, S, and a one-to-one correspondence, M, between C and S. The set S may be finite or infinite.

Consider the singleton sets $\{c_1\}$, $\{c_2\}$, ..., where each $c_i \in C$, and formats S_1, S_2, ... where each $S_i \in S$ and is determined by the correspondence $M(c_i)$. A variable format is defined as the union of the cartesian products of these singleton sets with their corresponding formats in S. Thus, the definition of a variable format is of the form

$$(\{c_1\} \times S_1) \cup (\{c_1\} \times S_2) \cup \ \ldots$$

This definition of a variable format may be abbreviated as $C \otimes S$ or $C \otimes \{S_1, S_2, \ldots\}$, with an implied one-to-one correspondence between C and S.

Although a format may be defined in terms of other formats, every format, after a finite number of substitutions, must be ultimately defined in terms of fields.

Two formats are equivalent if they can be made identical by applying the following rules to their definitions:

(i) $A \times B = B \times A$

(ii) $A \times (B \times C) = (A \times B) \times C$

(iii) $A \times (B \cup C) = (A \times B) \cup (A \times C)$

where A, B and C are formats. This concept is useful when fields in a format are rearranged to form a key (to be defined) while maintaining an equivalent format.

1.3 Records and Logical Files

A record is defined to be a member of a format; a fixed format record is defined to be a member of a fixed format, and a variable format record is defined to be a member of a variable format. In a variable format record, the value of the control field indicates the format of the rest of the record.

A logical file or a file is defined to be an empty set or a collection of records. Equivalently, a file may be defined to be a subset of a format. This definition does not allow two identical records in a file due to the nature of a set. From this viewpoint, two identical physical records stored in a computer system are not truly identical, because each record has an implicit field, its physical address, which makes the records distinct.

1.4 Keys

A primary key is a field or a format defined from a collection of fields of a given format and the value or values of the primary key uniquely identify each record in any file that is considered legitimate. That is, a primary key may exclude certain files or a format from consideration. A key is a field or a format defined from a collection of fields of a given format. The word "key" connotes the usage in defining a record or a collection of records in a file or a format by giving condition(s) on the field value(s) of the key. If K is a format consisting of only the fields in a key, then there exists a format L such that K × L is equivalent to any format that contains all the fields of the key.

1.5 Physical Files and File Organizations

A physical file is a representation of a logical file on a storage medium. A file organization is a physical file or a collection of physical fiels used to represent a logical file. In the representation process, each field value and record of the physical file is assigned a physical address and every record of the logical file corresponds to a record of the physical file.

A physical file may have a different format from that of the represented logical file, and a file organization may contain additional files to facilitate operations against the file. For example, physical address and record length (in bytes, words, etc.) may be introduced as

additional fields; a field whose values are variable-length may be redefined as the cartesian product of several fixed-length fields; an index file may be introduced as an additional file, as in index sequential files.

1.6 Operations on File Organization

The basic operations on a file organization are read, add, delete, and update, which involve single records. All other file operations can be described in terms of these basic operations.

The read operation makes available a particular record which satisfies a condition or determines whether such a record exists. In case a condition specifies more than one record in the file organization, all such records are considered to form a file with a certain ordering. We read the first record that satisfies the condition and then repeatedly read the next record until the read operation indicates no more records.

The add operation adds a given record to the file organization, possibly at a specific relative position.

The delete and update operations operate on a record which has just been read. If the operations are to be done on multiple records specified by a certain condition, a number of read operations, e.g., read first on the condition and read next on the condition, may be carried out before the delete or update operation. To delete a record means either to remove it from the file organization or to change it to a special record, a deleted record, that does not satisfy any conditions on the records.

2. EXAMPLES

A few examples are now given to illustrate the above definitions and concepts. The following fields are used in these examples.

I = the set of non-negative integers,

NAME = the set of all personal names,

A = the set of alphanumerical characters,

ADDRESS = the set of all addresses, and

CAR = the set of all car makes.

Example 1 Let E1 be the cartesian product of NAME and ADDRESS, i.e.

$$E1 = NAME \times ADDRESS$$

Note that E1 is a fixed format.

Let John Smith and Mary Dean be two instances of values of the field NAME. Similarly, let 113 Webster Street, 15640 Salina Street, and 3364 Main Ave., Apt. 3B be three instances of the field ADDRESS. Then the ordered pair (John Smith, 113 Webster Street) is an example of a record of the format E1. The set

{(John Smith, 113 Webster St.),
(Alexander Moore, 15640 Salina St.),
(Mary Dean, 3364 Main Ave., Apt. 3B)}

is an example of a file of format E1.

Example 2 Let format E2 be given by

$$E2 = I \otimes \{\{<e>\}, CAR_1, CAR_2, CAR_3, \ldots\},$$

where CAR_1 denotes CAR and CAR_n denotes the n-fold cartesian product of CAR. Note that E2 is a variable format and I is the control field. Each value from I in a record indicates the format of the record, i.e., the "n" of CAR_n.

Examples of records in this format are:

(0, <e>)

(1, Dodge)

(4, (Ford, Ford, Dodge, American))

Example 3 Let E3 be given by

$$E3 = NAME \times (C \otimes \{\{<e>\} , CAR\})$$

E3 is another example of a variable format where the control field C has exactly two members each indicating whether {<e>} or CAR is the relevant format for the rest of a record in E3. The following is a file in E3.

{(John Smith, (0, <e>)),

(Mary Dean, (1, Ford)),

(Alexander Moore, (1, Dodge))}

The field CAR in format E3 is sometimes called an optional field, since, in a sense, it may or may not exist for a record in E3.

Example 4 Let

$$E4 = NAME \times I \circledast \{S_0, S_1, S_2, \ldots\},$$

where S_i is a subset of CAR with i members and a member i of I corresponds to S_i, for i = 0, 1, 2, ...

This is another example of a variable format. Since a member of S_i is a file of the format CAR, each record of E4 contains a file of CAR which is called a subsumed file of the record. The value of the control field I indicates the number of records in the subsumed file.

3. SEQUENTIAL AND DIRECT FILES

3.1 Sequential Files

A sequential file is a physical file in which one record appears "after" another in some sense and this order is determined by the physical addresses of the records. Note that every sequential file, unless empty, has exactly a first record and a last record.

A physical sequential file is a sequential file in which a record A follows a record B if and only if A is stored after B, in the sense that no valid data is stored "between" A and B and the address of A is "greater" than B as determined by the characteristics of the storage medium.

A pointer sequential file is a sequential file whose ordering is represented by an added pointer field. As an example, consider a pointer sequential file of the format S = F × R, where R is the format

of the represented logical file and F is the set of all record addresses. In this example, the value of F in a record of S indicates the address where the next record is assigned. The value of F in the last record identifies the record as the last one in the file.

A variation of the above pointer sequential file has the format $F \times R_n$, where R_n is the n-fold cartesian product of R. Yet another variation has the format $F \times B \times R$, where B is the backward pointer field that contains the physical address of the previous record. The value of B in the first record indicates the record as the first.

Any physical file may be considered to be a sequential file, as every record has a physical address which determines the next and prior records.

In a sequential file, if record A folows record B, then the address of A is not known until B is examined either for its pointer field (pointer-sequential) or for the address of record B (physical sequential).

3.2 Operations on Sequential Files

The basic read operations are read the first record, and read the next record according to the ordering of the sequential file. Certain sequential files also allow the operations read the last record and read the prior record. To read a record satisfying a condition, the records from the first record on must be read one by one in the order specified by the sequential file, or, alternatively,

the records are read from the last record backward if possible.

It may not be possible to add a record to a physical sequential file if the added record must be between two specific records and there is no space between them. In this case a new physical file must be generated or the add operation fails. In a pointer sequential file, a new record is added between any two records by adjusting the pointers.

3.3 Ordered Sequential Files

It is possible to maintain a sequential file such that the next-prior relation of the records coincides with a condition on the values of a key. Such sequential files are said to be ordered. For instance, the value of a certain field K in a record A may be kept to be always smaller than or equal to the value of K in the next record of A (records in ascending sequence). When add, delete and update operations are performed on ordered sequential files, care must be taken to preserve the ordering condition.

3.4 Direct Files

A direct file is a physical file stored in such a way that (i) the range of all physical addresses is known without accessing any physical file, and (ii) given the address, each record may be found directly. The latter is a requirement on the storage medium.

Direct files are closely related to sequential files. As noted before, a direct file may be used as a sequential file, since the first record and the next record for each record are defined by the addresses

of the records. In fact, a direct file is also a physical sequential file ordered by the field physical address (implicit or explicit).

Reversely, sequential files may be used as direct files if (i) the set of all possible addresses is known, (ii) unused space always stores the representation of a deleted record, and (iii) the storage medium permits random access.

In a direct file, read is simple. To delete a record, either it is marked as deleted or its address is removed from the set of possible addresses. The add and update operations are straight forward.

3.5 More Complicated File Organizations

In a more complicated file organization, the records are partitioned and each partition is stored as a sequential file, a direct file or another file organization.

For example, consider the case when a record A must be added between two records B and C in an ordered physical sequential file, but there is no space between B and C. In order to avoid the regeneration of the entire file in such a situation, an overflow technique may be used. With such a technique, the records are partitioned into a number of subfiles, which are stored as pointer-sequential files. The first buckets of these subfiles are stored as physical sequential files in one area and the rest are stored in a separate area.

4. INDEX SEQUENTIAL FILES

4.1 Description of the File

An index sequential file consists of a data file and a hierarchy of index files. Each record in the data file corresponds to a record in the logical file which the index sequential file represents. The index files are supplementary files and are used to facilitate operations on the file.

To use the index sequential file organization technique, a key must be defined.

The data file may use any file organization technique and is partitioned into a number of data subfiles. An example is to store the data file as a physical sequentail file of the subfiles, where each data subfile has a fixed number of records. Another example is to store each subfile as a pointer-sequential file and to store the first buckets of the data subfiles as a physical sequential file, while keeping the records ordered by the key.

An index file for a file of format K × F is of format R × A, where R is the set of ranges of K and A is the set of physical addresses pointing at subfiles of the indexed file. The key range field R of an index record contains the range of the key values of the records in the subfile pointed at by the pointer field A.

An index file I may be created and maintained for a data file or for another index file J. In the latter case the R field of I contains the union of ranges of the records in a subfile of J. In

this way, an index sequential file may have a hierarchy of index files. We will say that the index file for the data file is at level 1, the index file for the index file at level 1 is at level 2, and so on. The index file at the greatest-numbered level is referred to as the top-level index.

The index subfiles are all ordered sequential files and are usually physical sequential files.

4.2. Operations on the File

To read a record satisfying a condition on the key, the top-level index is serially searched to locate the subfile of the lower-level index whose key value range contains a key value satisfying the condition. The same search is done at the still-lower level index and repeated until a data subfile is located. The data subfile is then searched serially to read a record satisfying the given condition.

To add a record, the record is inserted in the data subfile as determined by the index files. If the number of records in a data subfile is fixed, it may become necessary to create a new data subfile and hence cause the adding of a record into the index file at level 1, which may in turn cause the creation of a new index subfile and the addition of a record in the index file at level 2. This rippling effect may continue to the top-level index and finally cause an extra level of index file to be created.

To delete a record means to delete the record from the data file. This is done according to the file organization used for the data file.

If a data subfile becomes empty, it may be desirable to delete the corresponding index record at level 1, which may in turn cause a subfile at level 1 to become empty. Similar to adding a record, this rippling effect may extend to the top level and may cause the top level to disappear.

To update a record, if the key value is not changed, the record is read and updated. If the key value is changed, the original record is first deleted and then a new record reflecting the desired changes is added.

5. HASH FILES

5.1 Description of the File

A hash file has a hashing function that determines an address, the home address for any given key value and a key must be defined. All records that have the same home address form a collision set. Two records belonging to the same collision set may either contain identical key values or different key values that correspond to the same home address.

A hash file has a primary area and an overflow area. The latter may be a separate area of may coincide with the primary area. The home addresses generated by the hashing function point at buckets, the home buckets in the primary area. Each home bucket can only store a fixed number of records. Those records that cannot be stored in their home buckets are called overflow records and are stored in the overflow area.

A hashing algorithm determines a series of addresses in the overflow area based on a given key value. An overflow record is stored at one of these overflow addresses. Note that a record must be stored either at the home address or at one of the overflow addresses and that all these addresses may be predetermined from the key value of the record by the hashing function and the hashing algorithm.

To read a record, the direct file composed of buckets at this predetermined series of addresses is searched.

If a separate overflow area is used, the overflow buckets for the same collision set may be linked together as a pointer sequential file. In this case, the hashing algorithm is used only to allocate buckets in the overflow area.

If the overflow area coincides with the primary area and it is desired to use pointer sequential file organization, there must be a pointer for each record to link together all the records in the same collision set.

All hash files may be derived by varying the hashing function, the hashing algorithm, and the way the overflow records are stored and retrieved. For some examples of various hash files,(see Knuth (1973), Buchholtz (1963), Morris (1968)and Peterson(1957).

The hash file organization may be used for an index file, leaving the data file free to use any file organization.

5.2 Operations on the File

The hash file must be initialized to fill the primary area with deleted records before any operation can be performed on it.

To read a record with a given key, the buckets at the home address and overflow addresses are searched. Note that, if allowed by the storage device, it is possible to design the home and overflow addresses such that all the home and overflow buckets are searched simultaneously. To read a record satisfying an arbitrary condition requires the records to be read one by one, treating the physical file as a sequential file.

To add a record, we search for a deleted record among the buckets at the home address and the overflow addresses. If one is found, the record is stored there, possibly updating the pointers in a pointer sequential file; otherwise the add operation fails. There are various ways to limit the search for a non-existent record.

To delete a record, the found record is simply changed to a deleted record.

To update a record, the found record is simply changed as desired unless the key value is changed. In the latter case, the old record must be deleted and the new record added.

REFERENCES

BUCHHOLTZ 1963

Bucholtz, "File organization and addressing," IBM Systems Journal, June 1963, Vol. 2, pp. 86-111.

KNUTH 1973

Knuth, D.E., "The art of computer programming - fundamental algorithms," Addison-Wesley, Reading, Massachusetts, 1973.

MORRIS 1968

Morris, R., "Scatter storage techniques," Communications of ACM, Jan. 1968, Vol. 11, No. 1, pp. 35-38.

PETERSON 1957

Peterson, W.W., "Addressing for random-access storage," IBM Journal of Research and Development, Apr. 1957, Vol. 1, No. 2, pp. 130-146.

Section 11

CONCURRENCY IN HASHED FILE ACCESS

Leo H. Groner
Amrit L. Goel

Syracuse University

Abstract

This report presents a technique, Concurrent Hashed Overflow (CHO), for improving the performance of hashed file systems in terms of both access counts and storage efficiency. The technique involves the exploitation of the ability of many existing computer systems to access more than one unit of data concurrently. Algorithms implementing CHO are presented and an analytic model describing its performance is developed. The problem of optimizing the performance of a CHO file is formulated as a dynamic programming problem and is solved numerically. Tables are presented giving dynamic programming optimum stage allocations and next stage state. From these tables optimum CHO designs may be easily obtained.

TABLE OF CONTENTS

1. INTRODUCTION

In the design of hash coded files it is generally accepted that the trade-off between performance efficiency and storage efficiency is inevitable. This trade-off is investigated by Van der Pool [11, 12, 13]. However, there exists a third variable, moreover one that is frequently underutilized, that can be traded for improvements in both performance and storage utilization. This variable is the level of concurrent device utilization. In this report we try to exploit the ability of many existing computer systems to access data on more than one device concurrently. We present an algorithm which we call Concurrent Hashed Overflow (CHO) which may be used to yield improvement in hashed file systems in terms of both access time and storage efficiency.

To access a record using CHO, an address set consisting of the initial address and overflow addresses is generated by a set of hashing functions. When this set of addresses represents distinct devices, seeks to these addresses may be performed concurrently. Since more than one bucket is retrieved, the probability of success in obtaining the desired record in one access cycle is increased. Algorithms implementing CHO are presented and an analytic model of its performance is developed. Allocating a CHO file evenly between devices may not be an optimal policy. The problem of CHO file design for optimum performance is formulated as a dynamic programming problem of the form: determine storage allocations to minimize expected accesses for a given level of concurrency, load

factor, bucket size and file size. This problem is solved numerically and tables are presented giving a dynamic programming problem optimum state allocations and next stage state. These tables may be used in optimum CHO file design.

2. CONCURRENT HASHED OVERFLOW PROCEDURE

2.1 General Description

Consider a hash coded file consisting of a primary area into which records are hashed. Collision overflows from this area can be handled in a number of ways. Chaining and open addressing are among the more popular techniques. One technique that might be considered involves the use of a separate overflow file F_2 for overflows from the primary file F_1. F_2 may itself be organized as a hash coded file. Overflows from this file may be stored in a third overflow file F_3. For each hash coded file in the sequence $F_1, F_2, F_3, \ldots, F_i, \ldots$, we may associate a sequence of different hashing functions h_i which map each key k into a sequence of addresses a_i. This sequence represents all the possible addresses where the record may be found. The important point concerning this sequence is that, given the key, the entire sequence may be immediately calculated.

If the files F_i are each assigned to a separate device, I/O operations may be initiated immediately to each file. For devices such as the IBM 2314, disc seeks on different

units may be overlapped. For the IBM 3330, both seeks and portions of rotation latency may be overlapped. The portions of the file accesses which cannot be overlapped are the start I/Os and the final examination of record keys in main memory. Even transmission of records from the devices to main memory may be overlapped to the extent that the devices may be assigned to different channels. For the 2314 and 3330, representative times in milliseconds for various operations are given in Table 1. From this table we see that the bulk of the time spent in accessing records may be overlapped.

Since the performance consequence of collision overflow may be reduced by concurrent accesses to the overflow chain, the possibility of decreasing the storage requirements by increasing the file load factor presents itself. Although the number of file accesses may increase, the number of file accesses cycles, i.e., non-concurrent accesses, is decreased. It is the number of access cycles that is the primary determinant of response time.

2.2 Flowcharts

The flowcharts in Figs. 1 and 2 demonstrate the concepts discussed. In this example four levels of files, a primary and three levels of overflow, are used. Use of the terms FORK, JOINT and TERMINATE follows the terminology of Anderson [1].

In Fig. 1, the HASHREAD routine sets a switch EXISTS and sets indexes RI and RJ to point to the proper buffer in BUCKET where the sought record may be found. HASHREAD forks

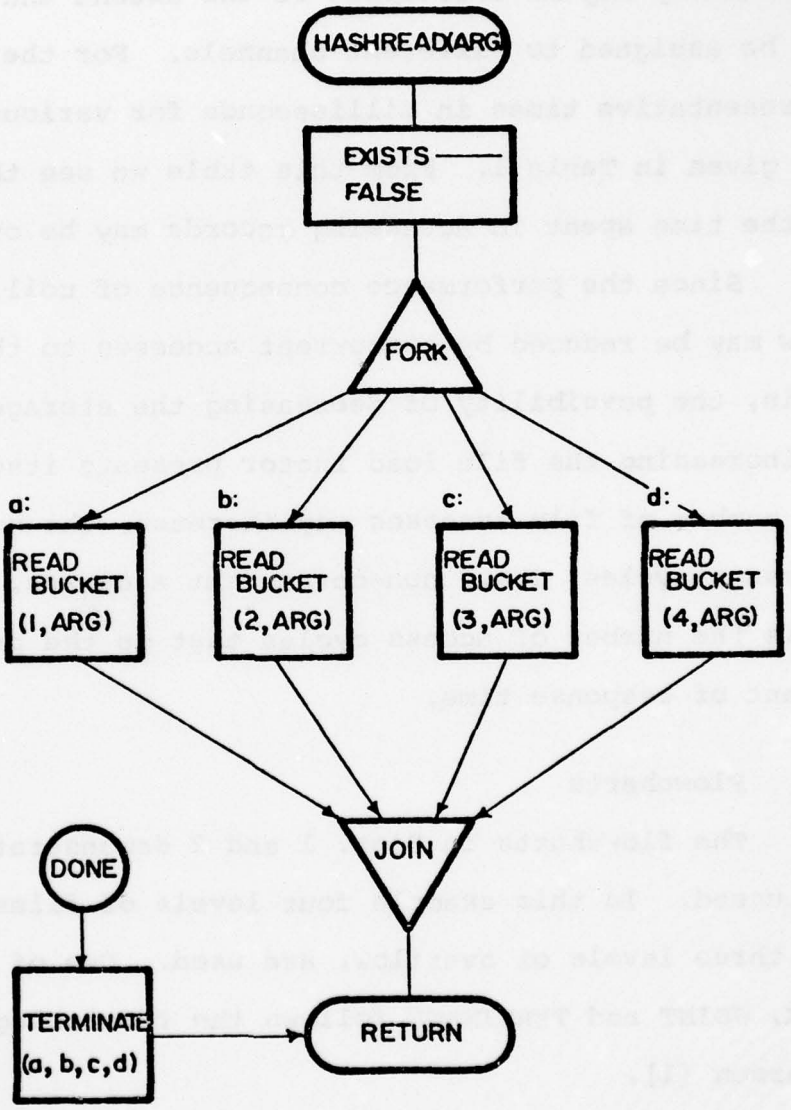


Fig. 1 The HASHREAD routine.

AD-A054 943 SYRACUSE UNIV N Y F/G 9/2
LARGE SCALE INFORMATION SYSTEMS. VOLUME II.(U)
MAR 78 F30602-74-C-0335
UNCLASSIFIED RADC-TR-78-43-VOL-2 NL

2 OF 3
AD
A054943

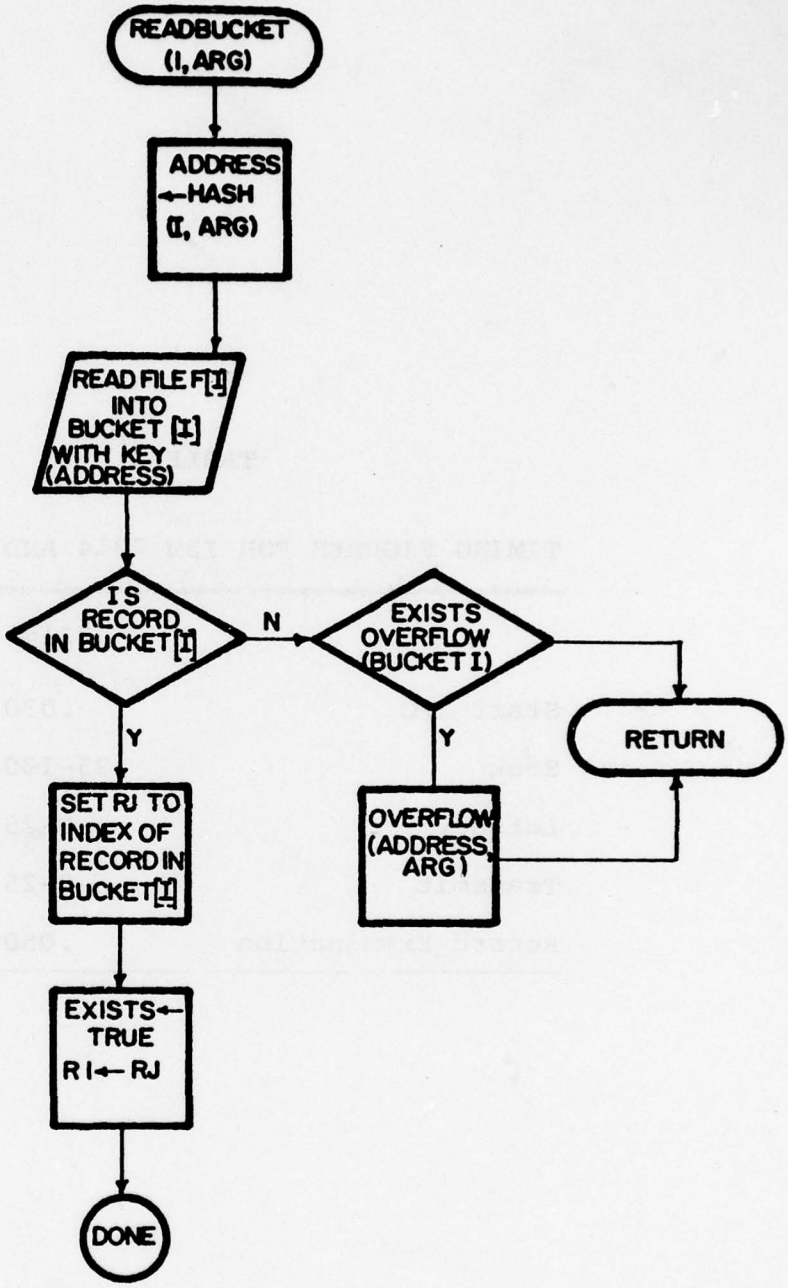


Fig. 2 The READBUCKET routine.

TABLE 1

TIMING FIGURES FOR IBM 2314 AND 3330 DASD

	2314	3330
Start I/O	.030	.030
Seek	25-130	10-55
Latency	0-25	0-16.7
Transmit	0-25	0-16.7
Record Examination	.050	.050

into four concurrent READBUCKET processes a, b, c, and d. HASHREAD may terminate if the condition DONE is signaled by the successful completion of any of its READBUCKET processes, or if all four READBUCKET processes have been completed, satisfying the JOIN statement.

In Fig. 2, READBUCKET computes the hash address associated with file F_i and tests whether the sought record exists in the bucket retrieved from that file. If the record is found, EXISTS, RI and RJ are set and the routine is exited by branching to DONE.

2.3 A PL/1 Implementation

The PL/1 program given in Fig. 3 is an implementation of the flowcharts in Figs. 1 and 2.

The MAIN1 routine defines the environment and calls the HASHREAD routine which sets EXIST and sets RI and RJ to point to the proper buffer in the BUCKET structure. HASHREAD initiates four READBUCKET subtasks which operate concurrently. If the record is found, an interrupt is generated by signaling DONE, transferring control to the OUT statement in HASHREAD.

```
MAIN1:  PROCEDURE OPTIONS(MAIN);
DCL(RI,RJ) FIXED BINARY(31), ARG CHAR(6);
DCL (F1,F2,F3,F4) FILE RECORD DIRECT ENVIRONMENT (REGIONAL(1));
DCL 1 BUCKET(4),2 RECORD(10),3 (KEY CHAR(6), DATA CHAR (24));
DCL EXISTS BIT(1)EV(4) EVENTS;
DCL 1 TARGET_RECORD,2 (KEY CHAR(6),DATA CHAR(24));
CALL HASHREAD(ARG);
IF EXISTS THEN TARGET_RECORD=BUCKET(RI).RECORD(RJ);
      ELSE /* WHATEVER */;
HASHREAD:PROC(ARG);
ON CONDITION(DONE) GOTO OUT;
EXISTS='0'B;
DO I=1 TO 4;
      CALL READBUCKET(I,ARG) TASK EVENT(EV(I));
END;
WAIT(EV(1),EV(2),EV(3),EV(4)) (4);/*WAIT FOR ALL OTHER
EVENTS TO COMPLETE */
OUT:RETURN;
DCL 1 FIXED BINARY(31) R(4) LABEL;
DCL TBUCKET CHAR(200),ADDRESS FIXED BINARY (31);
ADDRESS=HASH(I,ARG);
GOTO R(I);
R(1):READ FILE(F1)INTO(TBUCKET ) KEY(ADDRESS);
BUCKET(I)=TBUCKET;
GOTO TEST;
R(2):READ FILE(F2)INTO(TBUCKET ) KEY(ADDRESS);
BUCKET(I)=TBUCKET;
GOTO TEST;
R(3):READ FILE(F3)INTO(TBUCKET ) KEY(ADDRESS);
BICKET(I)=TBUCKET;
GOTO TEST;
R(4):READ FILE(F4)INTO(TBUCKET ) KEY(ADDRESS);
BUCKET(I)=TBUCKET;
TEST:DO J=1 TO 10;
```

Fig. 3 A PL/1 implementation (cont.)

```
      IF BUCKET(I).RECORD(J),KEY=ARG
      THEN DO R=I;RJ=J;EXISTS='1'B;
              SIGNAL CONDITION(DONE);
           END
      END;/* OF J LOOP */
IF (I=10) EXIST_OVERFLOW(BUCKET)(10))
THEN CALL OVERFLOW(ADDRESS,ARG);
/*PUTS OVERFLOW ADDRESS IN RI,RJ AND FILLS BUCKET (K)*/
END READBUCKET;
EXIST OVERFLOW:PROC(BUCKET) RETURNS(BIT(1));
/*DUMMY ROUTINE*/ RETURN('0'B);
END EXIST_OVERFLOW;
OVERFLOW:PROC(ADDRESS,ARGUMENT);
/*DUMMY ROUTINE*/
END OVERFLOW;
HASH;PROC(I,ARG) RETURNS(FIXED BINARY(31));
DCL I FIXED BINARY(31),ARG CHAR(6);
/* DUMMY */ RETURN(10);
END HASH;
END HASHREAD;
END MAIN1;
```

Fig. 3 A PL/1 implementation.

3. ANALYTICAL MODEL AND OPTIMIZATION METHODOLOGY

One way of formulating the problem of optimizing the design of Concurrent Hashed Overflow files is: minimize the extra accesses due to the kth stage overflow subject to an overall load factor L_0, bucket size s and a requirement for $r_0 = L_0 sB$ records to be hashed, where B is the number of buckets.

We are given $\sigma_0 = sB$ storage positions for records, i.e., slots. We allocate x_i slots to device i, $i = 0,1,\ ,k-1$. Following Van der Pool [11], the overflow records r_{i+1} from the (i+1)th stage satisfy:

$$r_{i+1} = r_i \frac{e^{-m_i} m_i^s}{s!} - \left(1 - \frac{m_i}{s}\right) \sum_{j=s}^{\infty} \frac{e^{-m_i} m_i^j}{j!} \qquad (1)$$

where $m_i = sr_i/x_i$. Slots remaining from the (i+1)th stage satisfy: $\sigma_{i+1} = \sigma_i - x_i$. (See Fig. 4) Letting $A_i = r_i/x_i$ and $L_i = r_i/\sigma_i$, the load factor after stage (i+1) is given by

$$L_{i+1} = \frac{A_i L_i}{A_i - L_i} e^{-sA_i} \frac{(sA_i)^s}{s!} - (1-A_i)\, e^{sA_i} - \sum_{j=0}^{s-1} \frac{(sA_i)^j}{j!} \qquad (2)$$

The stage to stage transition has now been expressed in terms of a single state variable L_i. Denoting the above expression by $t(L_i, A_i)$ we have $L_{i+1} = t(L_i, A_i)$. (See Fig. 5.) We note that R_i is the return from stage i and is a function of the state variable L_i and decision variable A_i.
Let $R_i(L_i, A_i) = L_{k-1}$ if $i = k-1$ and 0 otherwise,

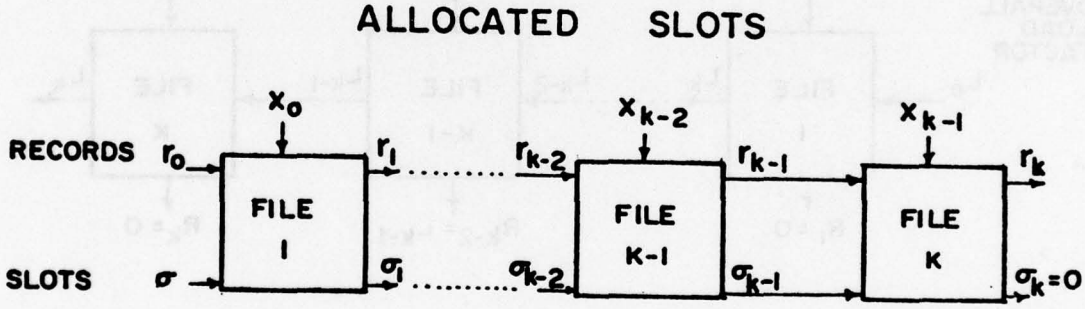


Fig. 4 Dynamic Programming formulation.

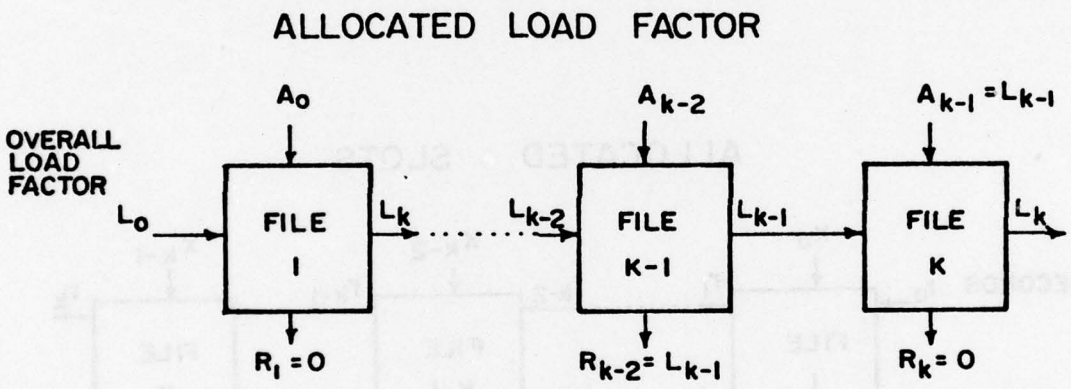


Fig. 5 Dynamic Programming reformulation.

$$f_i(L_i) = \min_{A_i \geq L_i} R_i(L_i,A_i) + f_{i+1}(L_{i+1}) \quad \text{for } i < k-2,$$

and $$f_{k-2}(L_{k-2}) = \min_{A_k \geq L_i} t(L_{k-2},A_{k-2}) \quad \text{for } i = k-2.$$

Then for $i < k-2$, $f_i(L_i) = \min_{A_i \geq L_i} \{f_{i+1}(t(L_i,A_k))\}$.

Since $f_{i+1}(\cdot)$ is monotonic, minimizing the expression above is equivalent to minimizing $t(L_i,A_i)$ over A_i subject to $A_i \geq$ $_i$. Using Lagrange multipliers we wish to minimize

$$F(L_i,A_i,\lambda_i,u_i) = t(L_i,A_i) + \lambda_i(A_i - L_i - u_i^2)$$

by setting:

$$\frac{\partial F}{\partial A_i} = \frac{dt}{dA_i} + \lambda_i = 0 \tag{3}$$

$$\frac{\partial F}{\partial \lambda_i} = A_i - L_i - u_i^2 = 0 \tag{4}$$

$$\frac{\partial F}{\partial u_i} = -2\lambda u_i = 0 \tag{5}$$

Eq. (5) holds if and only if $\lambda_i = 0$ or $u_i = 0$. Assuming $\lambda_i = 0$, A_i may be determined by solving $\frac{dt}{dA_i} = 0$ for a solution A_i^*. Alternatively, assuming $u_i = 0$ implies $A_i = L_i$. Then $A_i = \max\{A_i^*, L_i\}$. Dropping the subscripts we get

$$\frac{dt}{dA} = \frac{e^{-sA}L^2}{A-L}\left[\frac{s(sA)^{s-1}}{(s-1)!} + (A-1)\left[s\left(e^{-sA} - \sum_{j=0}^{s-2}\frac{(sA)^j}{j!}\right)\right]\right.$$

$$+ \left(e^{sA} - \sum_{j=0}^{s-1}\frac{(sA)^j}{j!} + \frac{1}{(A_i-L)} - s\right)\left[\frac{(sA)^s}{s!}\right.$$

$$\left.\left. + (A-1)\left[e^{sA} - \sum_{j=0}^{s-1}(sA)^j\right]\right]\right] = 0. \tag{6}$$

Let A^* be the solution to equation (6). Then $L_{i+1} = t(L_i, A_i^*)$ is minimum. The algorithm for generating the optimal allocations is therefore:

1. Initialize L_0, set $i = 0$, and set k equal to the number of stages.
2. Compute optimum A_i^* given L_i.
3. Set $L_{i+1} = t(L, A_i^*)$.
4. $i = i + 1$.
5. If $i < k-1$, then go to 2, else go to 6.
6. $A_{k-1}^* = L_{k-1}$.

Given the $A_0^*, \ldots, A_{k-1}^*$ we may use eq. (1) to determine the number of records in each stage, $r_0, \ldots, r_{k-1}$, and the slots allocated to them $x_0 \ldots x_{k-1}$, by $x_i = r_i / A_i^*$.

4. TABLES FOR DESIGN

The optimum allocated load factor A_i^* at different bucket sizes s and stage input load factors L_i is obtained by solving eq. (6) numerically. These values are given in Table 2 and Fig. 6 is a plot of the data in Table 2. Different curves are given for different values of s.

Table 3 is obtained by applying eq. (2) for the A_i^* values of Table 2. Fig. 7 is a plot of data from this table giving output load factors when optimum allocated load factors are used as a function of input load factors. The different curves represent different bucket sizes s.

5. NUMERICAL EXAMPLE

The following example may be illustrative. Suppose we are given a file of 100000 records with a bucket size 10 and an overall load factor 0.90. A primary file and two levels of overflow are to be used. They will be stored using the CHO technique on three separate devices. Thus the file will be stored in 11112 buckets allocated as given below. Overflow from File 3 is 490 records and these may be stored in File 3 using an OPEN addressing scheme.

TABLE 2

OPTIMUM ALLOCATED LOAD FACTOR A_i^* TO STATE (i+1).

	bucket size, s					
L_i	1	2	3	5	10	25
0.200	0.305	0.273	0.256	0.239	0.222	0.215
0.300	0.461	0.415	0.389	0.363	0.336	0.314
0.400	0.621	0.560	0.527	0.491	0.453	0.424
0.500	0.782	0.709	0.669	0.623	0.575	0.535
0.600	0.947	0.863	0.816	0.762	0.702	0.650
0.650	1.030	0.941	0.891	0.834	0.768	0.709
0.700	1.114	1.021	0.968	0.907	0.837	0.770
0.750	1.198	1.102	1.047	0.983	0.907	0.834
0.800	1.283	1.183	1.127	1.060	0.981	0.901
0.850	1.369	1.267	1.209	1.140	1.058	0.972
0.900	1.455	1.351	1.292	1.223	1.139	1.049
0.950	1.543	1.436	1.377	1.308	1.224	1.133
0.975	1.586	1.479	1.420	1.351	1.268	1.179

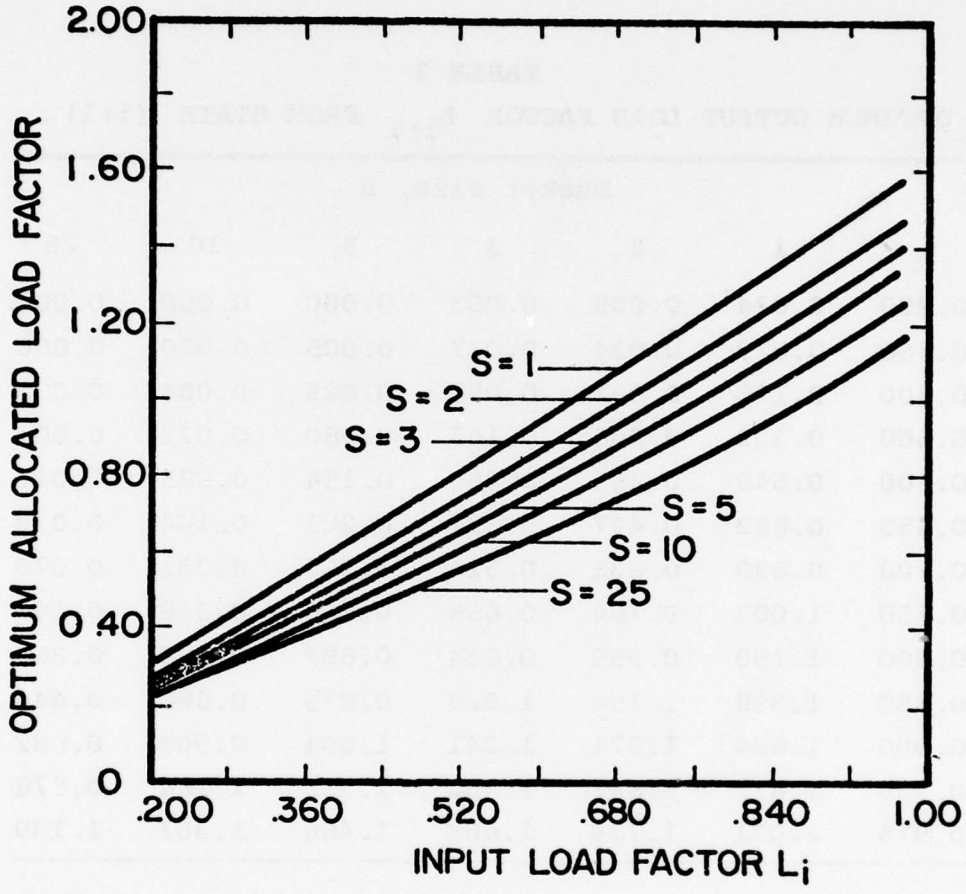


Fig. 6 Optimum allocated load factor A_i^* as a function of input load factor L_i.

TABLE 3

OPTIMUM OUTPUT LOAD FACTOR L_{i+1} FROM STATE (i+1).

	bucket size, s					
L_i	1	2	3	5	10	25
0.200	0.024	0.008	0.003	0.000	0.000	0.000
0.300	0.079	0.034	0.017	0.005	0.000	0.000
0.400	0.178	0.097	0.059	0.025	0.004	0.000
0.500	0.332	0.208	0.145	0.080	0.022	0.001
0.600	0.548	0.383	0.295	0.194	0.085	0.011
0.650	0.682	0.497	0.397	0.281	0.144	0.030
0.700	0.833	0.631	0.521	0.391	0.231	0.070
0.750	1.002	0.784	0.666	0.526	0.348	0.145
0.800	1.190	0.958	0.834	0.687	0.499	0.266
0.850	1.398	1.154	1.025	0.875	0.686	0.444
0.900	1.624	1.371	1.241	1.091	0.909	0.682
0.950	1.871	1.611	1.480	1.335	1.166	0.976
0.975	2.001	1.739	1.608	1.466	1.307	1.139

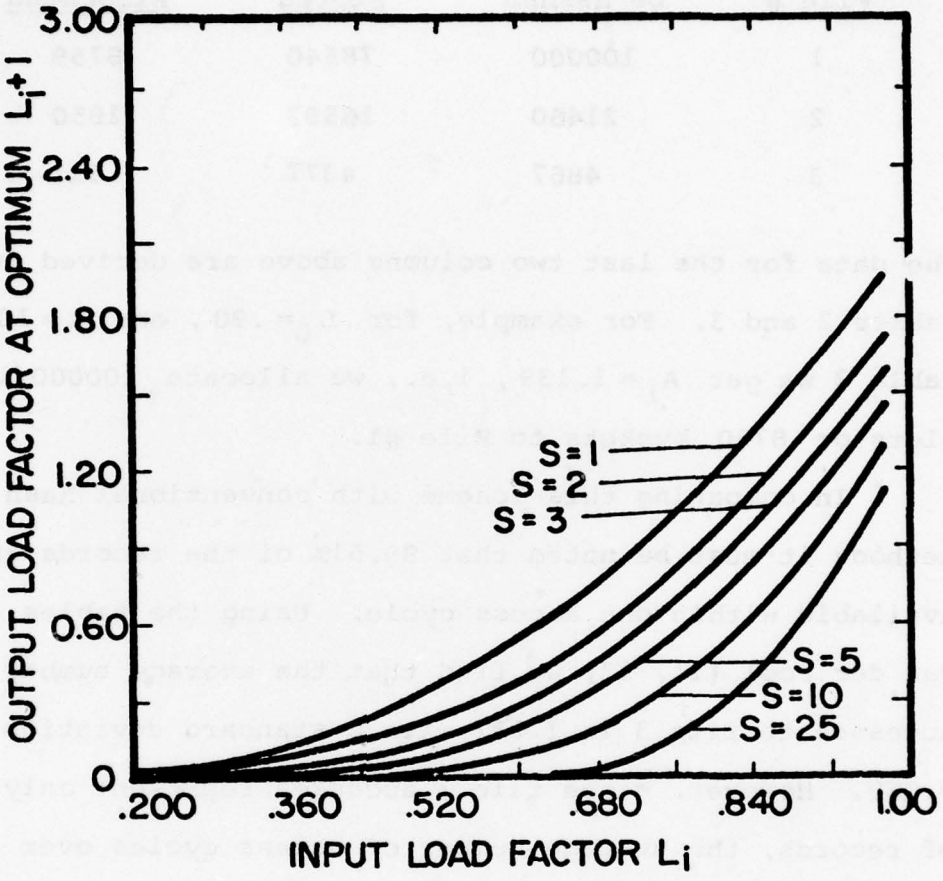


Fig. 7 Optimum output load factor L_{i+1} as a function of input load factor L_i.

File #	Records to be Hashed	Actually Stored	Buckets Allocated
1	100000	78540	8759
2	21460	16593	1850
3	4867	4377	511

The data for the last two columns above are derived from Tables 2 and 3. For example, for $L_0 = .90$, and $s = 10$ from Table 2 we get $A_i = 1.139$, i.e., we allocate 100000/1.139 slots or 8750 buckets to File #1.

In comparing this scheme with conventional hash coding methods it must be noted that 99.51% of the records are available within one access cycle. Using the tables in Van der Pool [11, 13] we find that the average number of accesses to file 3 is 1.837 with a standard deviation of 0.069. However, since file 3 accesses represent only 4.37% of records, the average number of access cycles over all three files is 1.037 with a standard deviation of 0.0145.

In Table 4 a comparison is provided between CHO, open addressing and chaining for the above numerical example.

Further improvement in performance may be achieved by either increasing the number of files or decreasing the load factor.

TABLE 4

COMPARISON OF TIMES FOR THREE SCHEMES

	Concurrent Access	Open Addressing	Chaining
Mean Access Cycles	1.0366	1.341	1.210
Standard Deviation	.0145	.011	N/-
# Concurrent Accesses	3	1	1
Load factor	.90	.90	.90 (in primary only) .834 (overall)
Bucket size	10	10	10

6. CONCLUSIONS AND REMARKS

In this report we have proposed a file access algorithm (CHO) based on concurrent utilization of multiple devices. We have developed a mathematical model of the performance of this algorithm and have demonstrated how this model may be used to optimize the CHO algorithm.

The algorithm developed and analyzed here is only one of a class of related algorithms that take advantage of existing concurrency in computer/channel/I/O device systems. Many airline reservations and message switching systems handle multiple transactions concurrently. The trandeoffs between improved throughput over many transactions at the expense of single transaction response time in those systems and opposite characteristics of the CHO algorithm described in this paper need to be examined. Furthermore, there are variations of this algorithm not yet analyzed. Nevertheless where single query response time is the major performance criterion it is clear that significant improvements may be effected by using CHO rather than searching the overflow records sequentially.

7. REFERENCES

[1] Anderson, J.P. Programming Structures for Parallel Processing, Communications of the ACM, Vol. 8, No. 12, December 1965, p. 786.

[2] Dijkstra, E.W. Solution of a Problem in Concurrent Programming Control, Communications of the ACM, Vol. 8, No. 9, September 1965, p. 569.

[3] Dijkstra, E.W. Structure of the 'THE' Multiprogramming System, Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 341-346.

[4] King, P.F. and A.J. Collmeyer. Data Base Sharing: An Efficient Mechanism for Supporting Concurrent Processes, Proceedings of the National Computer Conference, 1973, pp. 271-275.

[5] Morris, R. Scatter Storage Techniques, Communications of the ACM, Vol. 11, No. 1, January 1968, pp. 38-44.

[6] Olson, C.A. Random Access File Organization for Indirectly Addressed Records, Proceedings of the ACM National Conference, 1969, pp. 539-549.

[7] Peterson, W.W. Addressing for Random-Access Storage, IBM Journal of Research and Development, I, 1957, pp. 130-146.

[8] Schay, G. and W.G. Spruth. Analysis of File Addressing Method, Communications of the ACM, Vol. 5, 1962, pp. 459-462.

[9] Shoshani, A. and A.J. Bernstein. Synchronization in a Parallel-Accessed Data Base, Communications of the ACM, Vol. 12, No. 11, November 1969, pp. 604-607.

[10] Tainiter, M. Addressing for Random Access with Multiple Bucket Capacities, Journal of the ACM, X, July 1963, pp. 307-315.

[11] Van der Pool, J.A. Optimum Storage Allocation for Initial Loading of a File, IBM Journal of Research and Development, XVI, 6, November 1972, pp. 579-586.

[12] Van der Pool, J.A. Optimum Storage Allocation for a File in Steady State, IBM Journal of Research and Development, Vol. 17, No. 1, January 1973, pp. 27-38.

[13] Van der Pool, J.A. Optimum Storage Allocation for a File with Open Addressing, IBM Journal of Research and Development, Vol. 17, No. 2, March 1973, pp. 106-114.

[14] Webb, D.A. The Development and Application of an Evaluation Model for Hash Coding Systems, Ph.D. Dissertation, Syracuse University, August 1972.

Section 12

CASCADE HASHING

Yuan Liu
Amrit L. Goel

Syracuse University

Abstract

In this report we describe a new file organization, cascade hashing, which uses effectively sequential probes to search for a record. The optimization of cascade hashing file design is formulated as a dynamic programming problem which is solved using a new forward procedure. Some results on the comparison of cascade hashing with traditional hash file organization are also presented.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

A hash file is commonly considered the best when the average number of probes is minimized, where a probe is the operation of reading a bucket and examining its content to search for a particular record. This criterion is adequate for files stored on truly random access storage media, such as core memory, since the probe time is constant on such media. For files stored on disks, the processing time is a function of both the number of probes and the time for each probe. Therefore, even when the average number of probes is minimized to be close to one, it is possible to reduce the processing time further by reducing the probe time.

Basically, there are two kinds of probes on a disk storage device, random probes and sequential probes. A random probe consists of seeking to the cylinder, latency, and transmission. A sequential probe consists of the transmission only. Depending on the characteristics of disk and the length of the record, a random probe could be 10 or more times longer than a sequential probe. In this situation, the number of probes alone is not adequate for evaluating the performance of a hash file.

The cascade-hash file organization for disk files proposed here does not reduce the number of probes, but it guarantees that all the probes after the first one are effectively sequential probes. Therefore, the processing time is reduced. More importantly, for the same average processing time, a higher load factor may be used so that less storage overhead is required by the hash

file. With a slight variation, it is possible to do the probes in parallel to reduce the processing time further. Another advantage is that partial reorganization is possible.

2. THE CASCADE HASHING METHOD

2.1 The cascade-hashing concept

In a hash file, if a separate overflow area is used, any file organization may be used to store the overflow records, either treating all the overflow records as a whole or treating them in groups. Let A[1] denote the primary area and A[2] the separate overflow area. It is possible to store the overflow records from A[1] as a hash file, using A[2] as the primary area and A[3], a third area, as the overflow area. Again it is possible to store the overflow records from A[2] as a hash file using A[3] as the primary area and another area A[4] as the overflow area. This "cascade hashing" may continue for n stages so that area A[n] is the last overflow area.

Although any file organization may be used in A[n], it is perhaps best to keep it a sequential file for simplicity, since only a very small portion of the records are expected to be stored in this area.

2.2 Cascade hashing for disk files

To adapt the above cascade hashing method for disk files, we would like to search in A[2], A[3], ..., A[n] serially in such a way that there is no seeking (read/write assembly movement) nor any latency. The search in A[1] may still require seek and/or latency.

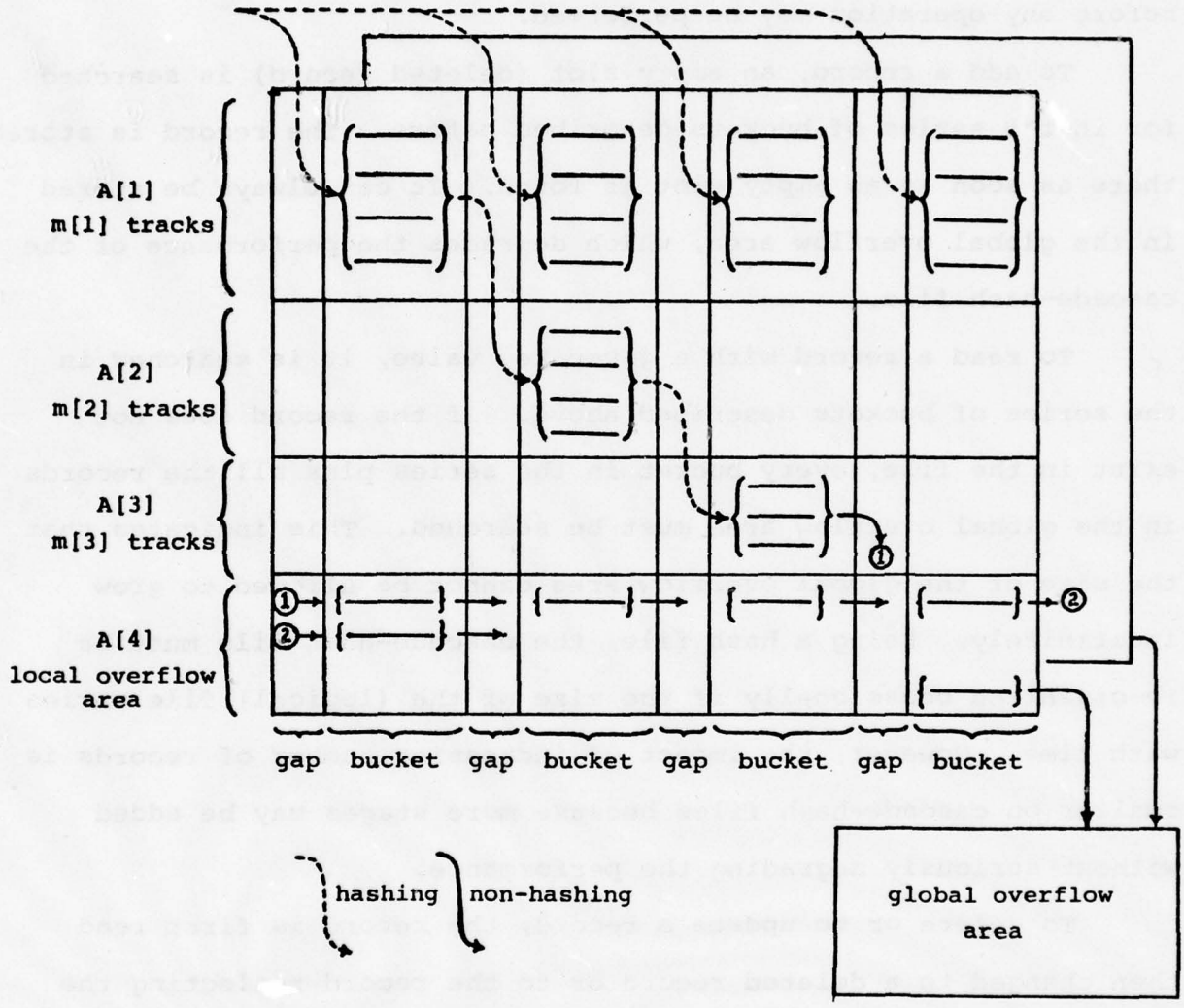
Basically, we arrange a chain of buckets for each key value so that the buckets are (1) in different areas, (2) on different

tracks of the same cylinder and (3) at consecutive angular positions. In most cases, the switch between tracks is so fast that it is completed before the next bucket is read. Therefore, the probes to these buckets are effectively sequential probes. Figure 1, explained below, shows how this is realized.

A number of cylinders are allocated to the cascade-hash file. For head-per-track disks, we consider that there is only one cylinder. A number of tracks on each cylinder are allocated to each area A[1], A[2],..., and A[n]. A record with a key value is hashed to a bucket in A[1] of a certain cylinder by a hashing function. Let the address be (C1,T1,B1), where C1 is the cylinder address, T1 the track address and B1 the angular position, expressed as a bucket number on a track. Note that the value T1 should make the bucket fall in area A[1].

The next bucket for overflow records from the home bucket is at C2 = C1, T2 = as determined by a second hashing function (must be in area A[2]), and B2 = B1+1. The bucket for overflow records from bucket (C2, T2, B2) is C3 = C2, T3 = as determined by a third hashing function (must be in area A[3]), and B3 = B2+1. Each of the subsequent buckets is determined in the same way of (1) keeping the same cylinder, (2) hashing to a track in the next area and (3) increasing the angular position by one bucket.

There may be a "global overflow area" A[n+1] in a separate area to store the records that may overflow from A[n] on each cylinder.



Shown is a 4-stage process in a cylinder with 4 buckets per track. Gaps are exaggerated. Hashing to this cylinder is not shown. Another cylinder is also shown in the background.

Figure 1 Cascade Hashing

2.3 Operations on a cascade-hash file

As in the case of any hash file, the entire area allocated to the cascade-hash file must be initialized with deleted records before any operation may be performed.

To add a record, an empty slot (deleted record) is searched for in the series of buckets described before. The record is stored there as soon as an empty slot is found. It can always be stored in the global overflow area, which degrades the performance of the cascade-hash file.

To read a record with a given key value, it is searched in the series of buckets described above. If the record does not exist in the file, every bucket in the series plus all the records in the global overflow area must be searched. This indicates that the size of the global overflow area cannot be allowed to grow indefinitely. Being a hash file, the cascade-hash file must be re-organized occasionally if the size of the (logical) file varies with time. However, the impact of increasing number of records is smaller on cascade-hash files because more stages may be added without seriously degrading the performance.

To delete or to update a record, the record is first read then changed to a deleted record or to the record reflecting the desired change. If the key value of a record is changed, then it must first be deleted and then added.

2.4 Considerations on implementation

The global overflow area A[n+1] may be stored using any file organization. In particular, records with the same home address may be chained together to limit the search for a non-existent record. If more tracks in each cylinder may be allocated, it is not difficult to add one more stage to the original cascade-hash file. Thus unlike the traditional hash file, partial reorganization is possible.

In order to limit the search for a non-existent record, one extra bit may be allocated for every bucket (or every record) to indicate the last bucket (or record) that has ever been stored any valid record. This bit is reset only by an add operation and is set only at initialization or reorganization time for every empty bucket (or slot). Note that this bit must not be changed by the delete or update operation after initialization.

If the areas A[1], A[2],..., A[n] are on separate packs, then parallel searching is possible, since all the addresses of the buckets may be pre-computed. This is a variation to the general cascade-hash file organization.

It is possible to use hashing with open addressing [PETERSON 1957] in the local overflow area A[n] in each cylinder instead of using the sequential file organization. This is not advisable, since it complicates the searching in this small area and it causes uneven use of the buckets in this area. The latter increases the possibility of overflowing from a local overflow area.

The records with the same home address (C1, T1, B1) may be chained together to limit the search for a non-existent record. The read time for an existent record is not affected because of latency. Note that chaining must be done at the record level, since each bucket must store records with different home addresses.

It is sometimes possible and more advantageous to let hardware, e.g., I/O controller or channel, do the searching.

The inter-record gaps on the track must be large enough so that examination of record key values in the current bucket, possible computation of the track address of next bucket, and switch to the new track can be finished in time before the next bucket arrives at the read/write head. If searching is controlled by software, there may be problems in maintaining control over the necessary I/O hardware.

Since n hashing functions with different ranges are used, it is advantageous to use a single algorithm that will give a hash address, using the range as an additional parameter to the key value.

3. OPTIMIZATION OF CASCADE-HASH FILES

3.1 Problem definition

The objective of this section is to formulate a problem for finding an optimal allocation of tracks of each cylinder to storage areas A[1], A[2], ..., A[n] for cascade hashing on disk storage devices. A design is considered optimal if it minimizes the average read time for an existent record. This criterion is equivalent to minimizing the average number of probes, as all the probes except the first one are sequential probes, whose probe times do not vary.

Let there be N_T records to be stored in an area of M_C cylinders on a disk storage device. Each cylinder has M_{TC} tracks, each capable of storing M_{BT} buckets or $M_{RT} = M_{BT} \cdot b$ records. The (over-all) load factor ρ is defined as $N_T/(M_C \cdot M_{TC} \cdot M_{RT})$, where the denominator, denoted by S, is the total number of records that can be stored in the whole cascade-hash file area. The above variables are all given at the time of design and cannot be varied.

Design Variable - bucket size: A bucket is a unit of storage that can be addressed, i.e., a bucket is read or written as one unit. A bucket may be chosen to contain one or more records. The bucket size, b, is the number of records that a bucket can contain.

Design Variable - number of stages: There are n stages, from A[1] to A[n]. The global overflow area should not contain any significant number of records for a well-designed cascade-hash

file, and, therefore, will be ignored for considerations of optimal designs.

Design Variables - area sizes: Within each cylinder, a fixed number of tracks are allocated to each area. Let m[1] tracks be allocated to A[1], m[2] to A[2], ..., and m[n] to A[n]. Note that

$$\sum_{i=1}^{n} m[i] = M_{TC} .$$

The read time is the sum of the first random probe time and the probe times of the subsequent sequential probes. For cascade-hash files, minimizing the read time is equivalent to minimizing the number of probes.

The average number of probes is computed by (1) multiplying the number of probes for each bucket by the number of records stored in it, (2) summing up these products for all buckets, and (3) dividing the sum by the total number of records.

The optimization problem then is:

given	ρ , M_{RT} , M_C , M
adjust	b , n , m[1] , m[2] ,..., m[n]
to minimize	average number of probes.

3.2 Notations

The following is a complete list of all the symbols used in this chapter.

a : average access time (unit = time to transmit a bucket) of records not in local overflow area.

A[i]: the area at stage i, $i = 1,2,\ldots,n$.

b : bucket size in records.

C[i]: $i = 0,1,\ldots,n$. number of tracks available for allocation to areas A[i+1] to A[n]. $C[0] = M_{TC}$. C[n] should be 0 for an optimal design.

b_T : bucket size in tracks. $b_T = b/M_{RT}$.

M : total number of tracks allocated to the cascade-hash file.

M_{BT} : number of buckets per track.

M_C : total number of cylinders allocated.

m[i]: $i = 1,2,\ldots,n$. number of tracks allocated to area A[i] in a cylinder.

M_{RT} : number of records per track.

M_{TC} : number of tracks allocated to the cascade-hash file on each cylinder.

N : average number of records stored on each cylinder.

n_s : number of stages, excluding the global overflow area.

N[i]: $i = 0,1,\ldots,n$. number of records to be hashed into area A[i+1] or overflowing from area A[i]. $N[0] = N$.

N_T : total number of records in the cascade-hash file.

Pov : the percentage of records stored in A[n], the local overflow area.

ρ : overall load factor, the ratio of total number of records to capacity of allocated area in records. $\rho = N/S$.

S : capacity of allocated area in records. $S = M_C \cdot M_{TC} \cdot M_{RT}$.

t[i]: $i = 1,2,\ldots,n$. read time contributed by records stored in area A[i].

t* : minimum average access time for given n_s and b (unit = revolution time). Its value is computed from the formula $t^* = aL + PovL[n-1+0.5(1/L + PovN)]$, where a and Pov may be looked up in the tables given in section 5.

v(n,m,b): number of overflow records when n reads are hashed into primary area of m buckets, each capable of storing b records.

3.3 Further conditions and observations on the design problem

The global overflow area is not included in the optimization problem, since, as we will see, the number of records in each local overflow area can be designed to be almost zero. In an implementation, this area must be provided in case some records are accidentally overflowed from a local overflow area.

The same bucket size is used in all the areas. This is required of areas A[1], A[2], ..., A[n-1] to achieve sequential probing. A smaller bucket size in the local overflow area A[n] may improve the processing time to some extent. However, its effect should be minimal, since only a small number of records are stored in a local overflow area.

Any integer value less than M_{RT} is possible for bucket size b and the number of buckets per track, M_{BT}, is given by the smallest integer that is larger than or equal to M_{RT}/b. On certain devices, only a portion of such values may be chosen. In case the optimal design requires an impossible bucket size, it is straightforward to find a next best value with a feasible bucket size.

- Theoretical Hashing Functions

Consider the key of a record to be processed as a random variable, denoted by K. A "theoretical hashing function" is defined as a transformation T on the random variable K such that T(K) is a discrete uniform random variable over the primary area, i.e., T(K) assumes each bucket address in the primary area with equal probability. The probability measure associated with K may represent probability that each key value appears in the file. In this case, the theoretical hashing function tends to distribute the records evenly among the buckets in the primary area, for most files that may happen.

However, for any given file, i.e., any given set of key values, a definite number of key values are hashed to each bucket. There is no randomness or probability involved. The assumption of such a theoretical hashing function is merely for providing a reference point among all possible hashing functions and for simplicity in analysis. Such a hashing function is assumed in most studies on hash files and hashing functions, e.g., [VANDERPOOL 1972].

It has been shown [LUM 1970] that a theoretical hashing function can be outperformed by a well-chosen hashing function for real files. Thus the performance of a hash file using a theoretical hashing function may be considered the worst case among all hash files with decent hashing functions.

- The Use of Means

If we consider an address generated by the hashing function as random variable, the number of records hashed to each cylinder is also a random variable. However, for any given file, the number of records hashed to a cylinder is a constant. A similar situation exists with the number of records overflowing from each stage of the cascade-hashing process. If we treat the numbers of overflow records as random variables, the optimal design obtained will be optimal among all possible files weighted by their probabilities of occurrence. Note that for any given file, it is often possible to obtain a design that performs better than such an optimal design.

In the following analysis we will assume a theoretical hashing function but we will use means for numbers of overflow records from the stages. The optimal designs so obtained may be considered as designs that one should usually choose without being given a specific file, and these designs are "typical" optimal designs. We can expect to find a better design if the characteristics of the given file do not change much during its life and such characteristics can be taken advantage of.

The mean number of overflow records, v, depends on the number of data records in the file, n, the number of buckets, m, and the bucket size, b. Since theoretical hashing function is used, the number of records N_B hashed to a bucket follows binomial distribution with parameters n and $1/m$, i.e.,

$$\text{Prob}(N_B = i) = C_i^n p^i (1-p)^{n-i} ,$$

where $p = 1/m$.

The number of overflow records from each bucket is given by, $N_v = 0$ if $N_B \leq b$ and $N_v = N_B - b$ if $N_B > b$; the mean number of overflow records from each bucket is thus given by

$$\begin{aligned}(1/m) \cdot v(n,m,b) &= \sum_{i=b+1}^{\infty} (i-b) C_i^n p^i (1-p)^{n-i} \\ &= \sum_{i=0}^{\infty} (i-b) C_i^n p^i (1-p)^{n-i} - \sum_{i=0}^{b} (i-b) C_i^n p^i (1-p)^{n-i} \\ &= n \cdot p - b + \sum_{i=0}^{b} (b-i) C_i^n p^i (1-p)^{n-i} .\end{aligned}$$

Therefore,

$$v(n,m,b) = m(n/m - b + \sum_{j=0}^{b-1} (b-j) C_j^n p^j (1-p)^{n-j}) .$$

Because the means are used, it is sufficient to consider only one cylinder, with $N = S \cdot \rho / M_C$ records to be stored in it. The mean number of overflow records is the same for every bucket in each area.

Further Assumptions:

It is assumed that each record in the file is accessed with equal probability when computing the mean read time.

The unit of time will be the revolution time of the disk storage device. The read time will not include the seek time and

latency time before reading the first bucket, since these times are the same for all cascade-hash files and hence is irrelevant for finding the optimal designs.

3.4 The optimization problem

The central problem of finding optimal designs is the optimization problem depicted in Figure 2. In the figure, the blocks represent transofrmations at stages and the arrows represent the input/output status of variables. Each output variable is a function of all the input variables of a block. The relationships among these variables are derives as shown in Figure 2.

By the assumptions made earlier, N records are to be stored in M_{TC} tracks. Let us consider one stage (i.e., area) at a time to estimate the total read time, which is N times the mean read time. The total read time is obtained by adding up the read times for the records stored in all the stages.

At stage 1, $N[0] = N$ records are to be hashed and $C[0] = M_{TC}$ tracks are available. If we allocate $m[1]$ tracks to area $A[1]$, then $C[1] = C[0]-m[1]$ tracks are available for stages 2 and on. Since there are $m[1] \cdot M_{BT}$ buckets in $A[1]$, the number of overflow records from this stage is $N[1] = v(N[0], m[1] \cdot M_{BT}, b)$, where v is the function given before. The total read time contributed by records stored in $A[1]$ is the product of the number of records, $N[0]-N[1]$, and the read time, $1/M_{BT}$ revolutions. Thus, $t[1] = (N[0]-N[1])/M_{BT}$.

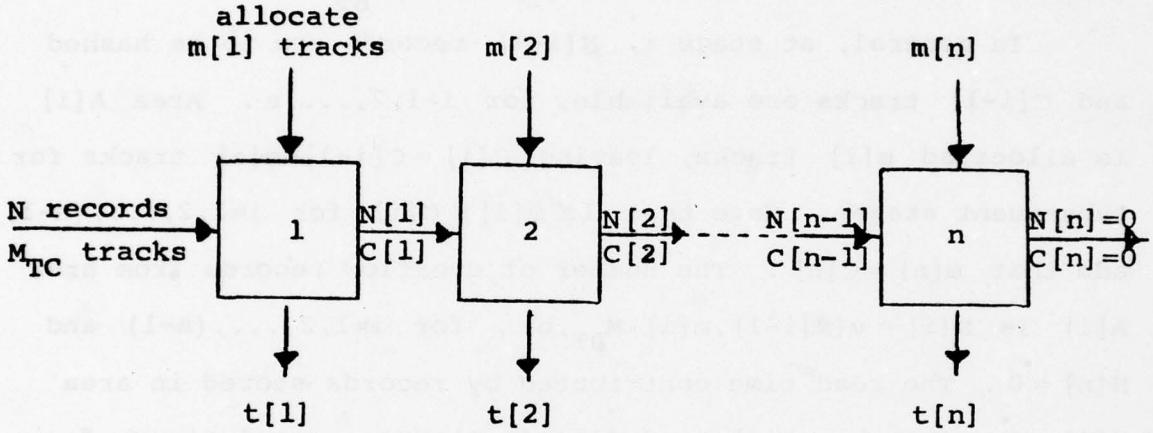


Figure 2 The Optimization Problem

At stage 2, there are $N[1]$ records to be hashed and $C[1]$ tracks available. If we allocate $m[2]$ tracks to area $A[2]$, then $C[2] = C[1]-m[2]$ tracks are left for subsequent stages. Since there are $m[2] \cdot M_{BT}$ buckets in $A[2]$, the number of overflow records from $A[2]$ is $N[2] = v(N[1], m[2] \cdot M_{BT}, b)$. The total read time contributed by records stored in $A[2]$ is the product of the number of records $N[1]-N[2]$ and the read time $2/M_{BT}$ revolutions, that is, $t[2] = 2 \cdot (N[1]-N[2])/M_{BT}$.

In general, at stage i, $N[i-1]$ records are to be hashed and $C[i-1]$ tracks are available, for $i=1,2,\ldots,n$. Area $A[i]$ is allocated $m[i]$ tracks, leaving $C[i] = C[i-1]-m[i]$ tracks for subsequent stages. Note that $1 \leq m[i] \leq C[i]$ for $i=1,2,\ldots,(n-1)$ and that $m[n] = C[n]$. The number of overflow records from area $A[i]$ is $N[i] = v(N[i-1], m[i] \cdot M_{BT}, b)$, for $i=1,2,\ldots,(n-1)$ and $N[n] = 0$. The read time contributed by records stored in area $A[i]$ is given by $t[i] = n \cdot (N[i]-N[i-1])/M_{BT}$ revolutions, for $i=1,2,\ldots,(n-1)$. The read time for a record in $A[n]$ is more involved.

The read time for a record in $A[n]$ is the sum of the time to search $(n-1)$ buckets sequentially, the rotational delay to reach the first bucket in $A[n]$ and the time to search in the sequential file in $A[n]$. See Figure 1. This is given approximately by $t[n] = N[n-1]\{(n-1) + M_{BT}/2 + (1+N[n-1])/(2 \cdot b)\}/M_{BT}$. The approximateness comes from the fact that information on disk storage device is read in units of buckets but this equation is based on the assumption that reads are in units of records. Since the

number of records in A[n] will be minimal for optimal cascade-hash file, this approximation will not cause significant error for the purpose of finding optimal designs.

If $N[i] > b \cdot C[i] \cdot M_{BT}$, the design is not feasible, since there is not enough storage left to store the overflow records from area i. In such cases we make the read time to be such a large quantity that the design will not be chosen.

Finally, the optimization problem, as depicted in Figure 2, is stated as

Given N, M_{TC}, and M_{RT}

adjust n, b, $m[1]$, $m[2]$, ..., and $m[n]$

to minimize $t = \sum_{i=1}^{n} t[i]$.

Note that M_{RT} serves to restrict the choices of b, namely, $1 \leq b \leq M_{RT}$ and $M_{RT} = b \cdot M_{BT}$, where M_{BT} must be an integer.

4. THE SOLUTION STRATEGY

4.1 The dynamic programming problem

For a given pair of (n,b), the above optimization problem may be formulated as a dynamic programming problem. Since the numbers of feasible values in n and b are small, it is possible to do an exhaustive search in the space of n and b, solving a dynamic programming problem for each pair of values of n and b.

Let N, M_{TC}, M_{RT}, n and b be given. We will formulate the dynamic programming problem and describe the solution strategy.

Let $f_i(N[i-1],C[i-1])$ be the minimum of $\sum_{j=1}^{n} t[j]$ by adjusting $m[i]$, $m[i+1]$, ..., $m[n]$. The Principle of Optimality stages $f_i(N[i-1], C[i-1]) = \underset{1 \le m[i] \le C[i-1]}{\text{MIN}} \{t[i] + f_{i+1}(N[i], C[i])\}$, $i=1,2,\ldots,(n-1)$, where $N[0] = N$, $C[0] = M_{TC}$, and $t[i]$ is a function of $m[i]$ and $N[i-1]$. For $i=n$, $f_n(N[n-1], C[n-1]) = t[n]$ and $m[n] = C[n]$.

The following is a summary of the relations derived before.

M_{BT} = the largest integer not exceeding M_{RT}/b,

$$v(n,m,b) = m\left(n/m - b + \sum_{j=0}^{b-1} (b-j) C_j^n p^j (1-p)^{n-j}\right), \quad p=1/m,$$

$N[i] = v(N[i-1], m[i]M_{BT}, b)$, for $i=1,2,\ldots,n$,

$C[i] = C[i-1]-m[i]$, for $i=1,2,\ldots,(n-1)$,

$t[i] =$ infinity if $N[i-1] > C[i-1]M_{BT} \cdot b$, for $i=1,2,\ldots,n$,

otherwise,

$t[i] = n \cdot (N[i]-N[i-1])/M_{BT}$, for $i=1,2,\ldots,(n-1)$, and

$t[n] = ((n-1) + 0.5M_{BT} + 0.5(1+N[n-1])/b) \cdot N[n-1])/M_{BT}$.

4.2 The traditional algorithm

Traditionally, a dynamic programming problem is solved "backwards" by computing $f_n(N[n-1])$ for all possible values of $N[n-1]$ and $C[n-1]$, and then $g_{n-1}(N[n-2], C[n-2])$ for all possible values of $N[n-2]$ and $C[n-2]$, and then $f_{n-2}(N[n-3],C[n-3])$ for all possible values of $N[n-3]$ and $C[n-3]$ using the just computed values for f_{n-1}, and so on, until $f_1(N[0])$ is obtained by using values of $f_2(N[1], C[1])$. One minimization problem has to be solved for computing each value of f_i.

In general, not all the f_i values computed will be used, making some of the computation efforts wasted. In the problem at hand, the values of $N[i]$ could be in the thousands and the value of $C[i]$ could be in the hundreds, thus hundreds of thousands of minimization problems have to be solved, although most of the values of $N[i]$ and $C[i]$ will not occur, especially at later stages (near n). In order to avoid unnecessary computation, an ingeneous scheme must be developed to estimate the possible values of $N[i]$ and $C[i]$ at each stage and solve f_i for these values only. If later a value of f_i is found to be needed but was not computed, one must be able to re-compute it, possibly triggering computation of f_{i+1} or even f of later stages.

However, there is a new forward algorithm which will automatically compute only the needed values for f_i, with very little extra complication.

4.3 The new forward algorithm

Basically, the new forward algorithm involves applying the Principle of Optimality stated above for $i=1,2,\ldots,(n-1)$, in that order.

To compute $f_1(N(0),C(0))$ a search method, such as binary search, is used to find the optimal value for $m[1]$. For a given $m[1]$, the values of $t[1]$, $N[1]$, and $C[1]$ are computed, then the computation of f_1 is suspended until $f_2(N[1],C[1])$ is computed. After $f_2(N[1],C[1])$ is computed, the computation of $f_1(N(0),C(0))$ continues until the optimal value of $m[1]$ is found by the search method.

To compute a value of $f_2(N[1],C[1])$, a search method is used to find the optimal value for $m[2]$. Similar to the previous procedure, an $f_3(N[2],C[2])$ is required for each trial value of $m[2]$, and the computation of $f_2(N[1],C[1])$ and the search algorithm is suspended until the particular $f_3(N[2],C[2])$ is computed.

The computation of $f_n(N[n-1],C[n-1])$ is straightforward.

To suspend a computation means storing all the variables and the point of suspension so that the computation may resume later as if the suspension did not occur.

In the above procedure, it may be advantageous to store the computed values of $f_i(N[i-1],C[i-1])$ in order to avoid duplicate computation in case the same value of $f_1(N[i-1],C[i-1])$ is required more than once.

The computer implementation of the new forward algorithm is straightforward in languages such as ALGOL and PL/1 which provide recursive procedures. Both the dynamic programming procedure and the minimization procedure must be recursive (i.e., being able to call itself).

4.4 Comparison of the two procedures

Since the new forward algorithm automatically avoids unnecessary computation of unused f_i values, it can be expected that the forward algorithm requires less computation. But then the storage requirement could be more.

As an example, assume there be 50 records to be stored and there be 20 tracks on one cylinder. If the backward algorithm is used, 1000 values of f_i need to be computed at each stage. This means 4,000 values in total for a 5-stage problem.

On the other hand, if the new forward algorithm is used, the number of f_i values computed is much less. Assume 4 tracks are allocated at each stage then about $(\log_2 20)(\log_2 16)$, $(\log_2 12)\cdots(\log_2 4)$, or about 380 f_i values will be computed. If the computed f_i values are stored, then the number will be even less.

In general, small number of stages, large N and M_{TC}, expensive minimization and relatively inexpensive storage favor the new forward algorithm. The programming cost for the two algorithms is about the same. After considering these factors, the new forward algorithm is chosen for finding optimal cascade-hash file designs.

4.5 The adopted algorithm

For given values of M_{RT}, t, n, N, and M_{TC} the following algorithm is used to compute the minimum access time and the optimal design, namely, optimal values ofr m[1], m[2], ..., m[n] .

We assume that the algorithm is implemented as a recursive procedure names DP(i,N,C) returning as its result the minimum access time for storing N records in C tracks at stage i (with n-i stages remaining). The procedure stores its findings as an entry (i,N,C,m*,t*) in a 5-column table, called f-table, where m* and t* are the optimal values.

We also assume available a minimization procedure MIN(D,OBJ,d*) which finds a d* such that $1 \leq d^* \leq D$ and OBJ(d) is minimized, where OBJ is a procedure which returns a result. The procedure MIN must be recursive. In our design problem, OBJ(d) gives as its result t[i] , the total access time of the records stored in stage i if d tracks are allocated to A[i] , and D is C[i] at stage i .

The procedure DP(i,N,C) is described below.

(1) If $N > b \cdot C \cdot M_{BT}$, then return with "infinity;" otherwise, continue.

(2) Search in the f-table for (i,N,C,-,-). If found, then return with t* ; otherwise, continue.

(3) Compute $t^* = MIN(C,OBJ,d)$, where $OBJ(m) = n \cdot (N-N')/M_{BT} +$ DP(i+1,N',C-m) and $N' = v(N, m \cdot M_{BT}, b)$. The golden-sectioning method is used in MIN.

(4) Set $m^* = d$, where d is given by MIN.

(5) Add an entry (i,N,C,m^*,t^*) to the f-table.

(6) Return with t^*.

To compute the minimum access time, initialize the f-table, compute $M_{BT} = M_{RT}/b$, where M_{BT} must be an integer, and then call $DP(i,N,M_{TC})$.

The above algorithm has been implemented in PL/1. These programs are used to compute the optimal cascade-hash designs.

5. OPTIMAL DESIGNS

The optimal designs for all practical values of n, b, and ρ have been obtained.

The average access time is composed of two terms, the first term reflecting the access time contributed by records in areas A[1], A[2], ..., A[n-1], and the second term reflecting the access time contributed by records in A[n], the local overflow area.

The first term is a product of the block length, $L = b/M_{RT}$, (block size in tracks) and a value which is a function of the load factor, $\rho = N/(M_{TC} \cdot M_{RT})$, block size b, and the number of stages, n, only, assuming the time unit to be the revolution time--the time to read one track of data.

The second term can be expressed as a function of n, b_T, S (total number of records that can be stored in all areas), and Pov (the percentage of records in the local overflow area), where Pov is a function of n and b only.

From the computed results, the minimum average access time obtained by solving a dynamic programming problem as described before may be expressed as $t^* = a \cdot b_T + Pov \cdot b_{T^*}(n-1 + 0.5(1/L+Pov \cdot N))$, where a and Pov are functions of ρ, b, and n only, and are given in Table 1.

The optimal designs for various values of R, b, and n are given in Table 2, assuming $M_C = 50$ tracks. If M_C is not 50, the tracks should be allocated proportionally.

Most of the results shown in the tables below have been obtained by assuming $S = 1000$ records, $M_{TC} = 50$ tracks, and $M_{RT} = 20$ records per track, and some of them have been checked with different values of N and M_{RT}.

(1) Load factor = 95%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1			2.04	18.8	1.80	27.3	1.41	40.0	1.00	60.4
2			1.92	7.6	1.66	14.1	1.36	25.2	1.00	47.8
4										
5	1.56*	0.0	1.51	1.3	1.52	3.3	1.30	10.3	1.00	31.2
10			1.30*	0.0	1.32	0.7	1.24	3.9	1.00	20.7
20			1.15*	0.0	1.15*	0.0	1.16	1.3	1.00	12.5

(2) Load factor = 90%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1			2.02	11.0	1.72	19.4	1.40	31.1	1.00	52.4
2			1.80	3.0	1.62	7.8	1.36	17.3	1.00	39.1
4			1.42	0.6	1.27	1.6	1.00	7.9	1.00	26.3
5			1.36*	0.0	1.35	1.1	1.26	5.7	1.00	23.4
10			1.18*	0.0	1.18*	0.0	1.18	1.3	1.00	14.4
20			1.09*	0.0	1.09*	0.0	1.10	0.3	1.00	7.9

(3) Load factor = 85%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1	2.10	3.7	1.96	6.9	1.69	13.9	1.39	25.5	1.00	46.5
2	1.62	0.7	1.64	1.2	1.53	4.1	1.32	12.9	1.00	32.7
4			1.32*	0.0	1.32	0.7	1.27	4.1	1.00	20.8
5					1.24	0.7	1.24	2.9	1.00	18.0
10			1.12*	0.0	1.12*	0.0	1.11	0.7	1.00	9.4
20			1.05*	0.0	1.05*	0.0	1.05*	0.0	1.00	4.5

TABLE 1 MINIMUM AVERAGE ACCESS TIME

(4) Load factor = 80%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1			1.87	4.5	1.66	10.1	1.35	21.5	1.00	42.0
2			1.51	0.6	1.47	2.1	1.29	9.1	1.00	28.8
4			1.25*	0.0	1.26	0.3	1.22	2.3	1.00	16.9
5			1.19*	0.0	1.19*	0.0	1.18	1.4	1.00	14.0
10					1.08*	0.0	1.08	0.4	1.00	6.9
20							1.03*	0.0	1.00	3.0

(5) Load factor = 75%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1			1.77	2.7	1.62	7.5	1.34	17.7	1.00	38.7
2			1.41	0.4	1.39	1.3	1.28	6.7	1.00	24.9
4					1.19*	0.0	1.18	1.3	1.00	14.1
5					1.14*	0.0	1.16	0.7	1.00	10.7
10							1.06*	0.0	1.00	4.8
20							1.02*	0.0	1.00	1.6

(6) Load factor = 70%

n	6		5		4		3		2	
b	a	Pov	a	Pov	a	Pov	a	Pov	a	Pov
1	1.74	0.6	1.70	1.7	1.58	5.3	1.33	15.0	1.00	35.3
2	1.33*	0.0	1.33*	0.0	1.32	0.9	1.24	4.7	1.00	21.6
4	1.14*	0.0	1.14*	0.0	1.14*	0.0	1.15	0.7	1.00	11.0
5			1.10*	0.0	1.10*	0.0	1.12	0.3	1.00	8.4
10							1.03*	0.0	1.00	3.4
20							1.01*	0.0	1.00	0.9

TABLE 1 MINIMUM AVERAGE ACCESS TIME

(1) Load factor = 95%
Total number of tracks = 50

b = 1

i	5	4	3	2	1
m[i]	16	12	8	5	9
		16	12	9	13
			18	13	18
				21	29

b = 2

i	5	4	3	2	1
m[i]	21	12	8	5	4
		22	13	8	7
			24	14	12
				27	23

t = 5

i	6	5	4	3	2	1
m[i]	30	13	5	1	1	0
		32	11	4	2	1
			28	14	6	2
				31	14	5
					35	15

b = 10

i	5	4	3	2	1
m[i]	39	8	2	1	0
		36	10	3	1
			36	12	2
				40	10

b = 20

i	5	4	3	2	1
m[i]		43	6	1	0
			41	8	1
				44	6

TABLE 2 OPTIMAL ALLOCATION OF TRACKS

(2) Load factor = 90%
Total number of tracks = 50

b = 1

i	5	4	3	2	1
m[i]	18	12	9	6	5
		19	14	8	9
			21	15	14
				26	24

b = 2

i	5	4	3	2	1
m[i]	23	13	8	4	2
		24	14	8	4
			26	16	8
				32	18

b = 4

i	5	4	3	2	1
m[i]	33	11	4	1	1
		32	12	5	1
			33	13	4
				38	12

b = 5

i	5	4	3	2	1
m[i]	36	10	3	1	0
		34	12	3	1
			39	10	1
				39	11

b = 10

i	5	4	3	2	1
m[i]	42	8	1	0	0
		41	8	1	0
			39	10	1
				43	7

b = 20

i	5	4	3	2	1
m[i]	46	3	1	0	0
		46	3	1	0
			44	5	1
				46	4

TABLE 2 OPTIMAL ALLOCATION OF TRACKS (continued)

(3) Load factor = 85%
Total number of tracks = 50

b = 1

i	6	5	4	3	2	1
	19	13	8	5	3	2
		19	13	9	6	1
m[i]			21	14	9	6
				23	16	11
					30	20

b = 2

i	6	5	4	3	2	1
	31	10	5	2	1	1
		27	13	6	3	1
m[i]			27	14	7	2
				29	15	6
					36	14

b = 4

i	5	4	3	2	1
	37	10	2	1	0
m[i]	.	35	11	3	1
			33	15	2
				41	9

b = 5

i	5	4	3	2	1
m[i]		39	8	2	1
			35	13	2
				42	8

b = 10

i	5	4	3	2	1
		45	4	1	0
m[i]			43	6	1
				46	4

b = 20

i	5	4	3	2	1
m[i]			47	3	0
				48	2

TABLE 2 Optimal Allocation of Tracks (continued)

(4) Load factor = 80%
Total number of tracks = 50

b = 1

i	5	4	3	2	1
m[i]	21	13	9	5	2
		22	15	9	4
			20	15	9
				33	17

b = 2

i	5	4	3	2	1
m[i]	31	11	5	2	1
		29	14	6	1
			31	15	4
				38	12

b = 4

i	5	4	3	2	1
m[i]	40	8	1	1	1
		37	10	2	1
			36	13	1
				43	7

b = 5

i	5	4	3	2	1
m[i]	41	7	2	0	0
		41	7	2	0
			38	11	1
				44	6

b = 10

i	5	4	3	2	1
m[i]		45	4	1	0
			45	4	1
				47	3

b = 20

i	5	4	3	2	1
m[i]			48	2	0
				48	2

TABLE 2 OPTIMAL ALLOCATION OF TRACKS (continued)

(5) Load factor = 75%
Total number of tracks = 50

b = 1

i	5	4	3	2	1
m[i]	24	12	8	5	1
		24	13	10	3
			27	16	7
				35	15

b = 2

i	5	4	3	2	1
m[i]	33	12	3	1	1
		32	12	5	1
			32	15	3
				40	10

b = 4

i	5	4	3	2	1
m[i]	40	9	1	0	0
		40	9	1	0
			38	11	1
				44	6

b = 5

i	5	4	3	2	1
m[i]		44	5	1	0
			39	10	1
				46	4

b = 10

i	5	4	3	2	1
m[i]			46	4	0
				48	2

b = 20

i	5	4	3	2	1
m[i]			49	1	0
				49	1

TABLE 2 OPTIMAL ALLOCATION OF TRACKS (continued)

(6) Load factor = 70%
Total number of tracks = 50

b = 1

i	6	5	4	3	2	1
m[i]	26	12	6	3	2	1
		25	12	8	4	1
			24	15	9	2
				28	16	6
					37	13

b = 2

i	6	5	4	3	2	1
m[i]	36	10	3	1	0	0
		36	10	3	1	0
			35	10	4	1
				34	14	2
					42	8

b = 4

i	6	5	4	3	2	1
m[i]	43	6	1	0	0	0
		43	6	1	0	0
			43	6	1	0
				39	10	1
					46	4

b = 5

i	6	5	4	3	2	1
m[i]	46	3	1	0	0	0
		46	3	1	0	0
			46	3	1	0
				41	8	1
					47	3

b = 10

i	5	4	3	2	1
m[i]			48	2	0
				48	2

b = 20

i	5	4	3	2	1
m[i]			49	1	0
				48	2

TABLE 2 OPTIMAL ALLOCATION OF TRACKS (continued)

6. DISCUSSION

1. Some general observations

The access time is the sum of two components, one contributed by records stored in areas other than the overflow area (regular component) and the other contributed by records stored in the overflow area (overflow component).

The regular component is proportional to the bucket size and a factor, a, that is a function of the design variables. The factor a varies much less than the overflow component does, for optimal designs.

The overflow component increases linearly with the total number of records and increases quadratically with the percentage of records in overflow area. Therefore, given a constant number of records in the file, if the number of records in the overflow area may be minimized to approach as closely as possible to zero, the overflow component is minimized. Two ways of doing this are to increase the bucket size and to increase the number of stages. Neither one increases the regular component significantly.

A fact to be remembered is that, at stage i, the number of records that overflow to next stage (i+1) is proportional to the number of records loaded, N(i-1), and decreases with a larger bucket size and/or a smaller local load factor, $N[i-1]/m[i] \cdot M_{RT}$.

2. The effect of number of stages

Since a fraction of the loaded records, N[i-1], is stored at every stage the number of records that overflow to next stage,

N[i] , decreases roughly geometrically with i . Therefore, the more stages there are the less records are stored in the overflow area and the smaller the overflow component is, hence the smaller the access time is. But when the number of overflow records has become effectively zero, the access time does not vary with the number of stages any more.

The results presented in the tables in previous sections indeed show that when the number of stages increases while keeping the bucket size constant, the optimal access time decreases until it reaches a minimum and stays there. This minimum is closer to the absolute lower bound for larger bucket sizes. This lower bound is the time to retrieve one bucket, (b/M_{RT}) revolutions.

The results also show that the optimal time reaches its minimum faster for greater bucket sizes. On the other hand, the optimal time reaches the minimum slower with a greater load factor. This is explained by the fact that a smaller percentage of records overflow to next stage if the bucket size is larger and/or the local load factor is smaller. For optimal designs, the local load factors are monotonic functions of the overall load factor.

3. The effect of bucket size

As described before, increasing the bucket size decreases the number of records in the overflow area, making the access time closer to the lower bound, b/M_{RT} . This generally decreases the access time. However, increasing the bucket size increases the

lower bound also. This is reflected in the data by the access time's first increasing then decreasing with the increasing bucket size, while keeping all other variables constant and the allocation optimal.

If the difference between the optimal access time and the absolute lower bound is plotted against the bucket size, a strictly decreasing curve results.

When there is no limitation on the number of stages, the optimal access time is very close to the lower bound, b/M_{RT}, which is minimized by the smallest possible bucket size.

4. Conclusion

When the searching is not interrupted by other activities on the system and there is no limitation on the number of stages, the bucket should be as small as possible. This is the case when the searching in the cascade-hash file may be done by one initiate of I/O command.

If the number of records in the file exceeds what the file organization is designed for and it is too expensive to reorganize the whole file, it is possible to reorganize only the overflow file into an additional stage or to reorganize only the last few stages, which are usually small.

REFERENCES

LUM 1970

Lum, V. Y., Ling, H., and Senko, M. E., "Analysis of a complex data management access method by simulation modeling," 1970 Fall Joint Computer Conference, pp. 211-222.

PETERSON 1957

Peterson, W. W., "Addressing for random-access storage," IBM Journal of Research and Development, Apr. 1957, Vol. 1, No. 2, pp. 130-146.

VANDERPOOL 1972

van der Pool, J. A., "Optimum storage allocation for initial loading of a file," IBM Journal of Research and Development Nov. 1972, pp. 579-586.

WHITE 1969

White, D. I., "Dynamic programming," Holden-Day, 1969.

Section 13

THE DESIGN AND IMPLEMENTATION OF APL-STARAN

John G. Marzolf

Syracuse University

Abstract

APL-STARAN is a dialect of APL designed to be more amenable to compilation than the standard version of APL. The principle differences include the use of name prefixes, the ability to accept a limited character set for denoting the primitive functions, some variations and restrictions on the use of the program-branching primitive, and the employment of a subset of APL functions with some additional I/O primitives. The reasons for each of these modifications are discussed in detail, as well as the implications for transportability between the two dialects. An implementation of APL-STARAN has been completed for the STARAN Associative Processor. An outline of this implementation as well as a user's guide and report of actual system performance are presented.

TABLE OF CONTENTS

LIST OF TABLES

SECTION 1: THE DESIGN OF APL-STARAN

APL as currently defined by the APL/360 implementation (see Refs. [1] and [2]) seems to be a natural candidate for a high-level language for SIMD-type parallel processors [3] - [5]. For a number of reasons, however, it does not appear that it can be conveniently compiled. For instance, the context-sensitive nature of a number of primitive symbols leaves their semantics ambiguous until execution time. Specifically, there are a group of primitive symbols which are used to denote two completely different functions depending on the number of arguments supplied with the symbol.

An example would be the cross (ordinary multiplication sign), which can be used monadically (i.e., with only a right argument) to represent the signum function, or it can be used dyadically (i.e., with both a left and right argument) to denote the multiplication function. In any given context the precise function intended can be determined by examining the source code to the left of the symbol. If there is some primitive scalar function on the left, then the cross has no left argument and hence is the signum function. If there is a right parenthesis or a number to the left of the cross, then it is known to be dyadic and thus is the multiplication function. For these cases the meaning is uniquely determined when the source code is entered. Such an arrangement allows compilation, provided one omits consideration of the memory management required for storage of the result.

The complication sets in when a name or identifier appears to the left of the cross. There is no way of knowing, prior to the actual execution of the statement, whether this name will represent (at execution time) a variable value or a monadic defined function call - and that makes all the difference. Another difficulty in compiling APL stems from the absence of declaration statements in the language. This makes memory management a very complicated process.

An obvious response to these difficulties is to run APL interpretatively. Not only does this solve the problems mentioned, but it also buys a number of little extras that are very convenient for interactive programming. On the other hand, it entails a certain loss of execution efficiency. This is counter-productive to the increased computational throughput expected from parallel processing.

The solution that is proposed here is to introduce a dialect of APL (to be called APL-STARAN) which can be more readily compiled, yet will remain as close as possible to the basic structure of standard APL. This permits parallel processing programs to be interactively written, tested and debugged on existing APL systems, and then transferred to the parallel processing systems for production use. It also provides the parallel processing programmers with a tested set of array oriented algorithms that have been developed by the APL programmers.

Design Goals and Their Rationale

There appear to be two solutions to the problem of using the same symbol to represent two different functions, namely (1) change one of the symbols, or (2) provide naming conventions which explicitly identify the class of a name used on the left of the ambiguous symbol. The first solution is the obvious one, but it is subject to the serious objection that it would be difficult to transport programs between the two dialects of APL. For this reason the second of the proposed solutions was chosen for APL-STARAN. This also solves the related problem of detecting the syntax error in an attempted assignment of a value to a niladic function name or a label.

All names in APL-STARAN are given alphabetic prefixes which specify the name-class of the object identified. There are five name-classes, and each is distinguished by a two letter prefix as shown in Table I. This means that all APL-STARAN names are a minimum of three characters long. The first two must be a proper prefix and the remaining characters may be any of the alphanumerics. Such a procedure causes these names to form a subset of APL names. Provided the names used in standard APL programs are chosen from that subset, all programs can be interchanged without alteration between the two dialects.

In order to free APL-STARAN from the need of having the special I/O devices necessary to generate the standard APL graphics, the APL-STARAN compiler has been designed to accept mnemonics in place of the "non-ASCII-like" APL symbols. There have already been a number of proposals for such a set of mnemonics (see Refs. [6] - [8]), but they generally employ a

TABLE I. NAME PREFIXES

Prefix	Name Class
VA	Variable
LA	Label
NF	Niladic function
MF	Monadic function
DF	Dyadic function

preceding escape character to differentiate the mnemonics from a similarly constructed APL name. In addition, the escape sequence is designed to be processed by the system input routine. This means that the conversion from the supplied mnemonic to the proper internal representation is made wherever the escape convention is encountered. In other words, if the escape sequence were used within a character literal, it would be converted by the system to the appropriate internal representation of the graphic for which it was the mnemonic. This is a suitable arrangement provided one is willing to allow the internal character count of some arrays to be different from that of their graphic representations. The conventions adopted for APL-STARAN have avoided this anomaly.

These conventions replace the "non-ASCII-like" APL graphics with a two letter mnemonic, which, because of the prefixing scheme chosen, cannot be confused with a name. These mnemonics are entered into the system precisely as the symbols for which they are the graphics (not as the symbols for which they are the mnemonics). The difference comes when the compiler encounters them. At this point they are translated into the proper execution code according to their mnemonic meaning. The mnemonics recognized by the compiler are shown in Table II. In cases where two mnemonics are shown for a given primitive, both have been provided merely to improve the readibility of the mnemonic source code, but they may be used interchangeably. The intended meaning is determined by the compiler following the same procedures it would use if the primitive graphic had been employed. Using this set of mnemonics, APL-STARAN programs may be entered from virtually any I/O device.

Three new primitive functions have been added to allow the full use of all available I/O devices. There is one input function (IN) which is monadic and takes a two component vector for its right argument. This argument specifies a dataset number which is the logical identifier of a physical device and a specific file. The assignment of a logical identifier to a given physical file is made by a command available in the DOS-11 operating system. The result generated by the input function is the value of the data object identified by its right argument.

There are two functions provided for generalized output; OB is the mnemonic for bare output and OT denotes terminated output. These functions are identical except for a final new-line character (carriage return plus

TABLE II. APL-STARAN SYMBOLS AND MNEMONICS (V1.1)

Primitive Scalar Functions

SYMBOL	MNEMONIC	DYADIC FUNCTION	MONADIC FUNCTION
+	+	Plus (Addition)	Identity (Numeric value)
-	-	Minus (Subtraction)	Arithmetic negation
×	TI,SG	Times (Multiplication)	Signum (Sign of)
÷	DV	Divide by	
⌈	MX,CE	Maximum	Ceiling
⌊	MN,FL	Minimum	Floor
|	RS,AB	Residue	Absolute value
*	*	Power	Exponential
⍟	LG	Base logarithm	Natural logarithm
<	<	Less than	
≤	LE	Less than or equal	
=	=	Equal	
≠	NE	Not equal	
≥	GE	Greater than or equal	
>	>	Greater than	
∧	AN	And	
∨	OR	Or	
⍲	NA	Nand (Not and)	
⍱	NR	Nor (Not or)	
~	NT		Not (Logical negation)

Primitive Mixed Functions

SYMBOL	MNEMONIC	DYADIC FUNCTION	MONADIC FUNCTION
⍴	RH	Restructure (Reshape)	Size (Shape)
⍳	IO	Least index of	Index generate
,	,	Last axis catenate	Ravel
,[]	,[]	Catenate or Laminate	

Special Functions and Symbols

SYMBOL	MNEMONIC	FUNCTION OR MEANING
←	IS	Assignment
→	GO	Go to (branch or jump)
	IF	Conditional Branch (used with GO)
[;]	[;]	Index selector
()	()	Grouping symbols
'	'	Quote (character string delimiter)
¯	^	Hi-minus (negative number indicator)
/	/	Vector reduction
⌶	IB	I-Beam (system dependent functions)
	IN	Monadic input (from file system)
⍞	NI	Niladic input (quote-quad input)
⍎	EX	Execute (evaluate)
⍞	EX NI	Evaluated input (quad input without prompt)
⍞←	OT	Terminated output (monadic or dyadic)
⍞←	OB	Bare output (monadic or dyadic)
∇	DL	Del (function delimiter)
;	;	Local variable separator
:	:	Label separator

line feed) that is supplied by the system at the end of each OT output request. Both functions are essentially dyadic (with conventions for eliding the left argument). The left argument specifies a logical dataset (similar to the input function) and the right argument provides the data object to be output. Like other APL primitives, the output functions generate a result, which in this case is the value of the right argument. In other words, ignoring the physical output produced, these functions act as if they were the identity function. An alternate way of viewing these two functions is as a specialized case of the assignment function where the left argument is the logical name of a shared variable [9].

The problem of memory management in the absence of dimension declarations is solved by subterfuge. Although subsequent versions of APL-STARAN may wish to attack this problem directly, the present version solves it by a get-space routine that does the job at execution time. This means that there is not a unique correspondence between the variable names and their physical memory addresses. Data values are located through a data descriptor table. Once some experience is gained with this implementation it will be possible to determine if a better solution to the problem is demanded.

There has been an improvement in the efficiency of the branching mechanism supplied with standard APL. Since the whole question of a proper set of control structures for APL is under investigation (see, for example, Refs. [10] and [11]) the arrangement described here should be seen as only an interim solution. Although the GOTO is not eliminated (indeed, it remains the kernel of the whole control system) it is made subject to a number of restrictions designed to provide more efficient execution. As a by-product these restrictions significantly improve one's ability to trace the flow of control through a program.

There are only three types of syntax allowed when using the branch arrow. It may be used

(1) with a single label, (including the null label LA),

(2) with an indexed label list, or

(3) with a single label followed by the mnemonic IF, followed by any APL-STARAN expression that evaluates to a 1 or 0.

This means that the target(s) of any branch is (are) always visible at the source level. Although this is more restrictive than the ordinary APL usage, it avoids the possibility of a variable name or function name completely obscuring the destination of the branch.

The indexed label list mentioned in the second type of branch is constructed by simply juxtaposing a number of labels as if they were the elements of a numeric vector, and then indexing this string. This is the format required by the compiler, but it is not permitted in standard APL. Nonetheless, its semantic equivalent may be easily obtained in APL by catenating the labels, enclosing them in parentheses and indexing the resulting expression. This multiple branch provides the "case selection" capability used by structured programming languages to execute one of a number of alternate program blocks.

The third type of branch uses a new primitive (IF) which is nothing more than the compression function with its arguments reversed, together with the restriction that its left argument must be a single label. Although this function is not available in standard APL as a primitive, it is easily introduced as a user defined function. Indeed, there are no real transportability problems between the two dialects because of these branching restrictions. All that is required is that these restrictions also be followed in the standard APL programs.

Basic Structure of the Implementation

An implementation of APL-STARAN has been completed for the STARAN Associative Processor at Rome Air Development Center. The compiler is itself written in APL-STARAN. It communicates with the I/O devices through a supervisor module written in MACRO-11 assembly code. This module runs on a PDP-11 which acts as the sequential control unit for STARAN. The supervisor module uses the basic I/O facilities for the DOS-11 batch operating sysyem, and presents the compiler with a source program in a form which is basically ASCII.

The compiler translates the source program into an object module in a form called QCODE. If the source program is a function definition, then the object module is merely stored in the control memory. Otherwise the compiler acts as a load-and-go system, deleting the object module when execution is complete.

QCODE is the APL-STARAN execution code, and is essentially a form of threaded code [12]. The QCODE machine is a stack oriented virtual machine composed of two main modules, one running on the sequential processor (SP) and the other on the associative processor (AP). There is also an interface module to manage the inter-processor communications via the external function logic and shared memory (which is actually AP control memory that is accessible to SP).

The system attempts to overlap as much as possible of the SP and AP execution. The SP module fetches the object code, decodes it, performs the necessary houskeeping, and makes a call to AP for an array operation. While AP is staisfying this request, SP proceeds to the next object code.

SECTION 2: HOW TO USE APL-STARAN

APL-STARAN is a dialect of a subset of APL.SV. Section 2 shows how to submit an APL-STARAN job, and sections 3 and 4 explain how to program in APL-STARAN.

This manual is based on Version 1.1 of APL-STARAN and is intended for use with the RADCAP Staran computer. Access to this facility is through the RADC Multics system, and jobs submitted in this fashion do not permit APL-STARAN to be used interactively. It is necessary to first prepare a complete APL "terminal session" by entering each APL statement as a separate line in a Multics segment named APL.IN. This segment may be prepared using the Multics editor "edm", or other equivalent editor. When this is completed the job is sent to STARAN by using the Multics command "staran" to enqueue the following dos-input (-di) segment:

```
$JOB [ "YOUR OWN UIC" ]
$RUN SDM
#WA
#X
#LD DK1:SARREX /NG
#TSCM
$RUN DK1:APL
$FINISH
```

The APL output generated by STARAN will be found in the Multics segment APL.OUT, where each APL input statement will be retyped with an indentation of 6 spaces, followed by the appropriate APL response. A sample of each step of this procedure may be found on the next page.

NOTE: In all the examples of actual terminal input/output that follow, the terminal used required that the symbols [,], and ^ be entered and received as ¢<, ¢>, and ¬ respectively.

```
edm apl.di
Segment not found.
Input.
$JOB ¢<64,46¢>
$RUN SDM
¢#WA
¢#X
¢#LD DK1:SARREX/NG
¢#TSCM
$RUN DK1:APL
$FINISH
.
Edit.
w
q
r 2120 0.397 6.064 191

edm APL.IN
Segment not found.
Input.
3+5
4 TI 6
7 - 2
.
Edit.
w
q
r 2121 0.266 1.026 54

staran -di apl.di
1 request submitted, 2 already queued.
r 2121 0.392 10.374 108


from Staran.SysDaemon: staran request processed: apl.di

pr APL.OUT 1
APL-STARAN  V1.1
08-OCT-75
22:14:43

      3+5
 8.00000E00
      4 TI 6
 2.40000E01
      7 - 2
 5.00000E00


r 2259 0.150 3.684 47
```

SECTION 3: THE BASIC APL-STARAN STATEMENTS

There are two main categories of mathematical functions in APL-STARAN:

(1) A fundamental set of basic arithmetic and logic operations, the so-called Primitive Scalar Functions, and

(2) A set of array manipulating operations known as the Primitive Mixed Functions.

In addition to these two main categories there are a number of specialized operations which are not conveniently grouped. Table II is a concise summary of all these operations, together with the symbols and mnemonics used to denote these functions.

There are also two modes of system operation for APL-STARAN:

(1) Execution Mode in which each APL statement is compiled and executed immediately after being read from the system input file (APL.IN) The result of this statement execution is then output to the APL.OUT file before the next input statement is read.

(2) Function Definition Mode in which a number of statements are successively read in, then compiled and stored for later execution.

The execution mode is the normal mode in the sense that the system is initially in this mode and remains in it until the function delimiter (DL) is encountered in the system input. This section of the manual deals with the execution mode. Section 4 explains the function definition mode and the procedures for entering and leaving that mode.

Basic data formats. There are two basic data types, namely, character and numeric. The logical data type is a subset of the numeric data type (1 = "true", and 0 = "false"). If a primitive function requires logical data for its operands, the system will automatically convert to a 1, any argument whose absolute value is greater than or equal to 0.5. If a primitive function requires numeric data and is supplied instead with character data, the system will automatically convert the character data to their ASCII code values.

Numeric data may be entered in the exponential format, e.g., 1.23E6 may be entered in place of 1230000. Regardless of how the data is entered, only the six most significant digits of any number will be displayed on output, and the internal format used for the storage of numeric data will maintain only

24 bits of significance. In fact, this is rounded to 21 bits for all relational tests. The absolute value of any exponent may not exceed 38.

Negative numbers are denoted by prefixing them with the ASCII circumflex symbol, or the PL/1 logical NOT (¬) if a 2741 terminal is being used. Vectors of both the numeric and character data types may also be entered. Numeric vectors are formed by simply juxtaposing their components, using spaces to separate each of the elements. Character vectors are entered by enclosing the character string in quotes ('). The quote symbol may also be embedded within such a character string by using two adjacent quotes.

Primitive Scalar Functions. The basic arithmetic and logic functions may be divided into two classes:

(1) Dyadic functions which require two arguments, one placed on each side of the symbol used to denote the function, and

(2) Monadic functions which have only one argument, placed to the right of the function symbol.

In APL-STARAN there are 19 dyadic and 9 monadic functions in the primitve scalar category. These are all listed in Table II. The meaning of each function should be reasonably well known, or may be determined by experimental use. If a precise definition is desired, any standard APL text may be consulted.

Although these functions are defined in terms of operating on scalar arguments, they may also be used with vectors, or even higher dimensional arrays. If the function is monadic, the effect of having an array as an argument would be to produce a result of the same size and shape as this argument. Each element of the result would be obtained by applying that particular function to the corresponding element of the argument.

For the dyadic functions there are two distinct situations which can occur. The determining factor is whether the left and right argument arrays are conformable or not (that is, of the same size and shape). If both argument arrays are conformable, then the result will conform to this size and shape. Its elements will be the result of applying this function to the argument arrays on an element-by-element basis in a manner similar to the monadic case.

If the arguments are not conformable, then there are two further cases which can occur. The first case arises when one of the arguments is a scalar or a single element array. In this case it will be "extended" to be conformable

with the other argument, and the result will be generated as if both arguments had been conformable. If neither argument is a single element, then a LENGTH or RANK error occurs. This will not be detected by the APL-STARAN system. Rather, the left argument will be forced to be conformable with the right argument, using "garbage" from the workspace, if necessary, to fill out the left argument. The result will thus be conformable with the right argument.

Assignment Statements. Except for "branching" statements (which occur in defined functions, and will not be discussed until Section 4), all statements produce a result of some sort which is generally written into the APL.OUT file. Instead of outputting this result, however, it may be stored in the workspace for future calculations. The symbol to denote the storing operation is IS, and statements of this type are usually referred to as "specification" or "assignment" statements; they specify a variable name to hold the result generated. To use such stored quantities one merely uses the variable name which is "holding" them, whether they be single numbers, or vectors, or general arrays. The rule for forming variable names is that they must begin with the letters VA followed by a sequence of 1 to 3 other letters or numbers. Consider the following example of an APL.OUT file.

```
      VAX IS 4+5
      VAX
 9.00000E00
      2+VAX
 1.10000E01
      VAY IS 2 3 ¯5
      ¯2 TI VAY
¯4.00000E00   ¯6.00000E00    1.00000E01
      VAY
 2.00000E00    3.00000E00   ¯5.00000E00
```

Line (1) illustrates the assignment operation. The indentation in line (2) indicates that there was no output as the result of performing the calculation in line (1); the system was waiting for another input. Calling for the variable in line (2) outputs its expected value in line (3). The variable may also be used in calculations as shown in lines (4) and (5). Lines (6) through (10) illustrate the storing of a vector along with the "extension" of a scalar to be conformable with a 3 element vector as required by the times (multiplication)

function.

The general rule on when output occurs and when it does not, is that it always occurs except when the operation leftmost in the statement is the assignment function. The only exception to this is with the use of the output function. A detailed discussion of this function will appear below. It is also worth observing that if any quantity (either a number or a literal string) is entered alone as an entire APL statement in itself, it becomes the result value generated by the statement, and as such will be output to the APL.OUT file, as shown in the following example.

```
      23
2.300000E01
      'ALPHABETIC DATA'
ALPHABETIC DATA
```

It must also be noted that any number of operations may be specified in one APL statement. This includes assignment and I/O operations as well as regular mathematical operations. If more than one operation is specified in a given statement, the order of operations is strictly from right to left unless parentheses are used to group operations into some other sequence. This is an important point since it differs from other programming languages which generally employ a more complicated order (usually involving some hierarchy of operations) and working from left to right. In APL all operations are equal in determining the order of execution. In the absence of parentheses, the first operation to be performed is the rightmost. If this is not the intended order of execution, parentheses may be used freely (including parentheses within parentheses) to group the operations as desired. If parentheses are used, they must be used in pairs, and everything within a given pair will be evaluated prior to its being used as a quantity in some other operation external to that pair. The following output example illustrates these points.

```
      1+2+3+4
1.00000E01
      (2 TI 3)+(4 TI 5)
2.60000E01
      (5-2)-6
¯3.00000E00
      5-2-6
9.00000E00
```

The only line that should require explanation is line (7). It is important to remember the right to left rule. Do not read this line as 5 subtract 2 gives 3, subtract 6 gives¯3. That's the way to read line (5). Because of the right to left rule, line (7) should be read as 2 subtract 6 gives¯4, and 5 subtract¯4 gives 9. Another way of looking at it is that every symbol operates on everything to the right of it, up to the first parenthesis which encloses it (assuming it's enclosed in parentheses).

Primitive Mixed Functions. Arrays are classed in accordance with their dimensionality. A vector is a one-dimensional array, as opposed to a matrix which is a two-dimensional array. In accordance with this classification a scalar should be considered as an array of zero-dimension. The dimensionality of an array is usually called its rank or shape.

The elements of an array are the individual numbers in a numerical array, or each separate character in a literal array. These elements are identified or distinguished from each other by their indices. An index is a number which locates the element relative to the origin or beginning of the array. For instance, in a vector a single index for each element suffices to locate it relative to the first element in the vector. The first element may be given the index 0 or 1 depending on the index origin setting, as described under "system dependent functions". A matrix would require two indices for each element; one to identify the row in which it appears, the other to identify the column. Rows and columns are numbered starting at either 0 or 1 depending on the index origin. Higher dimensional arrays are allowed, and are indexed similarly.

The size of a specific array is given by the number of elements in the array. (The size of a matrix would require two numbers to specify it.) As an example, the size of the vector 1 7 9 3 is 4, while the size of the

matrix

```
2   5   14
1   3    6
```

would be 2 3 (the row index is always given first). It is also possible to have a vector of size 0. This means there are no elements in the array, and it is referred to as an empty vector.

The Reshape (or Restructure) operation is a dyadic function which restructures the right operand (which can be any array) to have the size and shape specified by the left operand. This operation may be used to make arrays of any size or shape. If there are not enough elements in the original array, they are used over again, starting from the beginning. If there are too many elements, the unused ones are dropped. Example:

```
     VAX IS 6 RH 2
     VAX
2.00000E00    2.00000E00    2.00000E00    2.00000E00
  2.00000E00    2.00000E00
     VAY IS 8 RH 1 2 3 4
     VAY
1.00000E00    2.00000E00    3.00000E00    4.00000E00
  1.00000E00    2.00000E00    3.00000E00    4.00000E00
     VAZ IS 3 RH 1 2 3 4 5 6
     VAZ
1.00000E00    2.00000E00    3.00000E00
     3 4 RH VAZ
1.00000E00    2.00000E00    3.00000E00    1.00000E00
2.00000E00    3.00000E00    1.00000E00    2.00000E00
3.00000E00    1.00000E00    2.00000E00    3.00000E00
     2 1 RH VAY
1.00000E00
2.00000E00
```

The size operation is a monadic function which yields the size of the operand (any array) on its right. The size operation always produces a vector for its result. The size of a single number is an empty vector, whereas the size of a vector is itself a vector of size 1. Thus, the size operation applied to the size of any array A (i.e., the size of the size of A) yields the shape, or rank, of the array, (i.e., its dimensionality).

Example:

```
      VAX IS 2 4 RH 2 5 ¯9
      VAX
2.00000E00    5.00000E00   ¯9.00000E00    2.00000E00
5.00000E00   ¯9.00000E00    2.00000E00    5.00000E00
      RH VAX
2.00000E00    4.00000E00
      RH 1 2 3 4 5 6 5 4 3 2 1
1.10000E01
```

The Indexing operation is somewhat special in the way it handles its operands. The function performed by the operation is the selection of specific elements in any array by the specification of the indices of the selected elements. The notation for indexing is the enclosure within brackets of the desired indices, which may themselves be in the form of an array. The brackets must be placed immediately to the right of the array to be indexed. Two indices are needed to index a matrix, and they are separated from each other by a semi-colon. If either one of the indices is omitted from a matrix index, the entire row or column specified by the other index is assumed to be referenced. The case of higher dimensional arrays is handled analogously. Example:

```
      VAX IS 2 4 RH 1 2 3 4 5 6 7 8
      VAX
1.00000E00    2.00000E00    3.00000E00    4.00000E00
5.00000E00    6.00000E00    7.00000E00    8.00000E00
      VAX¢<1;1¢>
1.00000E00
      VAX¢<2;3¢>
7.00000E00
      VAX¢<2;¢>
5.00000E00    6.00000E00    7.00000E00    8.00000E00
      VAX¢<;3¢>
3.00000E00    7.00000E00
      VAX¢<2 1 2;3 3 1 4¢>
7.00000E00    7.00000E00    5.00000E00    8.00000E00
3.00000E00    3.00000E00    1.00000E00    4.00000E00
7.00000E00    7.00000E00    5.00000E00    8.00000E00
```

The Index Generator is a monadic function requiring a positive integer for its operand. It generates a vector of successive integers of the size specified by the operand. If the operand is zero, it generates an empty vector. The first integer in the vector will be the index origin. Example:

```
      ⍳12
1.00000E00    2.00000E00    3.00000E00    4.00000E00
  5.00000E00    6.00000E00    7.00000E00    8.00000E00
  9.00000E00    1.00000E01    1.10000E01    1.20000E01
      ⍳0
```

The Least-Index-Of function (dyadic ⍳) requires a vector for its left argument but will accept an array of any size or shape for its right argument. It generates a result array which is conformable with the right argument. Each element in the result is the lowest index of the left argument where the corresponding element of the right argument is to be found. Any elements in the right argument which do not occur in left argument will generate a least index which is one greater than the highest index of the left argument. Example:

```
      3 1 4 2 5 ⍳ 0 2 7
4.00000E00    6.00000E00
```

The Ravel function (monadic comma) manipulates the size and shape of any argument to make it into a vector by "stringing out" the elements in row-major order. In its dyadic form the comma denotes the catenate function. For this function the arguments must be either scalars or vectors. The function then creates a longer vector by appending the right argument to the right end of the left argument. If the arguments are higher dimensional arrays they should be conformable (or one should be a scalar or single element array). The right argument will then be appended to left argument along the last dimension. For instance, in the case of two matrices, the result array will have the same number of rows as the arguments, but the number of columns (i.e., last dimension) will be equal to the sum of the number of columns in the left and right arguments. The columns on the right side of the result will be those from the right argument. The situation for other higher dimensional arrays will be analogous. If the arguments are not conformable, they will be made conformable as described

under "Primitive Scalar Functions". Examples will be found following the discussion below.

This function also has an "indexed" form created by enclosing an integer index in brackets and inserting it between the comma and right argument. The function of the index is to select the dimension along which the catenation should take place. Legal index values are thus limited to the number of dimensions in the arguments of the function. If the index is not an integer the function becomes the laminate function. Lamination creates an array of rank one higher than the arguments by inserting a new dimension of size 2 above the dimension indicated by the floor of the fractional function index. The procedure for forming the result is best illustrated through the various examples given below.

```
      2 1,3 4
2.00000E00    1.00000E00    3.00000E00    4.00000E00
      9,2 2RH 10 4
9.00000E00    1.00000E00    2.00000E00
9.00000E00    3.00000E00    4.00000E00
      VAX IS 2 3 RH 10 6
      VAX,¢<1¢>9
1.00000E00    2.00000E00    3.00000E00
4.00000E00    5.00000E00    6.00000E00
9.00000E00    9.00000E00    9.00000E00
      VAX,¢<2¢>9
1.00000E00    2.00000E00    3.00000E00    9.00000E00
4.00000E00    5.00000E00    6.00000E00    9.00000E00
      9,¢<2.5¢>VAX
9.00000E00    1.00000E00
9.00000E00    2.00000E00
9.00000E00    3.00000E00

9.00000E00    4.00000E00
9.00000E00    5.00000E00
9.00000E00    6.00000E00
      9,¢<1.5¢>VAX
9.00000E00    9.00000E00    9.00000E00
1.00000E00    2.00000E00    3.00000E00

9.00000E00    9.00000E00    9.00000E00
4.00000E00    5.00000E00    6.00000E00
```

The Reduction operator (/) in APL-STARAN requires a vector for its right (and only) "real" argument, but it also requires one of the 6 functions +, TI, MX, MN, AN, OR as a "pseudo" left argument. It generates a scalar which is the result that would be obtained if the specified function were inserted between each of the elements of the vector. In other works "+/VAX" yields the sum of all the elements in the vector named VAX.

The Execute function (EX) requires a character vector as its right argument and executes (evaluates) the character string as if it had been entered through system input APL.IN. It works by generating an internal call to the APL-STARAN compiler, and produces the same results (including user function definition) as would normally be obtained from the compiler.

System Dependent Functions. The I-beam primitive (IB) may be used monadically to accomplish a number of system oriented functions. In all cases the argument must be either a scalar or a two element vector. The allowed scalar arguments and their respective system functions are:

IB 21 returns the elapsed CPU time in 1/60 seconds.

IB 48 returns the index origin.

IB 50 returns the digits control setting (number of digits used for output).

IB 51 returns the width control setting (number of output characters per line).

IB 64 terminates APL-STARAN.

The use of a two element vector for the argument allows a number of system controls to be given values. In each case the first element of the vector determines the precise system function to be performed, and the second element is the data necessary to accomplish it. (A more complete discussion of the I/O operations will be found in the next section). The implemented function are:

IB 36 N ($1 \leq N \leq 7$) open dataset "N" for I/O.

IB 37 N ($1 \leq N \leq 7$) closes dataset "N" for I/O.

IB 48 N ($0 \leq N \leq 1$) sets index origin to N.

IB 50 N ($1 \leq N \leq 6$) sets digits control to N.

IB 51 N ($32 \leq N$) sets width control to N.

Input/Output Functions. If the output of a vector will not fit on one line, subsequent lines will be used and be indented to indicate the continuation of the vector. An empty vector is output as a blank line. In matrix

output the columns will be aligned, provided each row fits on a single line. If a row requires more than one line, subsequent lines will be indented to indicate the continuation of the row. Column alignment will hold for corresponding lines of each row.

APL-STARAN uses 8 datasets for input and 8 for output. For purposes of communicating with STARAN's operating system these datasets are named DIØ, DI1, ..., DI7 for input, and DOØ, DO1, ..., DO7 for output. The system input is dataset DIØ and is internally assigned to a Multics segment named APL.IN. The system output is dataset DOØ which is assigned to a Multics segment APL.OUT. Both of these assignments may be altered by using the DOS-11 monitor command $ASSIGN before executing the DOS command $RUN DK1:APL. The details of this procedure are described in Digital Equipment Corporation's Manual "Disk Operating System Monitor Programmer's Handbook" (DEC-11-OMONA-A-D). The other datasets are internally assigned in input/output pairs to a corresponding Multics segment named APLN.DAT where N is a number from 1 to 7 that matches the dataset number. For example, DI2 and DO2 are both assigned to the Multics segment APL2.DAT. These assignments may likewise be changed by the DOS command $ASSIGN.

The system input and output datasets are automatically opened and closed by the initiation and termination of APL-STARAN. The other datasets must be opened by the appropriate I-Beam function before they can be used for input or output. The number of datasets that may be simultaneously open is limited by the available buffer space. In order not to lose any data that has been written into these datasets they must be closed with the appropriate I-Beam before terminating APL-STARAN. When datasets 1 through 7 are written, the new data is <u>appended</u> to what is already in the file. In the case of dataset 0, the file APL.OUT is erased and rewritten each time APL-STARAN is initiated.

The monadic function IN uses its argument (between 0 and 7) to identify the dataset from which to read. Character data is read serially from this dataset, starting from the present read "position" up to the first terminator encountered (e.g. carriage return). The new read position is set following this terminator. The character vector thus read becomes the value returned by this function. There is a <u>niladic</u> form of this function (niladic means <u>no</u> arguments) which is equivalent to IN Ø. In other words, NI has no argument and reads from the system input file APL.IN.

There are two forms of the output function. Bare output (OB) differs from terminated output (OT) in that the latter function supplies a terminator (new-line character) to the end of the output string. The left argument identifies the dataset where the output is to go. If it is Ø it may be omitted, which is equivalent to the function becoming a monadic function. In both cases the right argument is the data to be output.

SECTION 4: DEFINING APL-STARAN FUNCTIONS

The system supplies a function definition operation (DL) which enables the user to organize any specific sequence of basic arithmetic and/or array processing operations into a defined function. This defined function will be identified by a function name which must be formed as described below. Once the function is defined it may be executed by using its name in an APL statement.

Defined functions may be divided into the same two groups used to classify the fundamental system-supplied functions, namely, monadic and dyadic. In addition there is a third classification available for defined functions called niladic - no operands. Functions in this classification may be considered as stand-alone programs which employ I/O operations to input their operands and output their results. The present discussion begins with this case. The procedure for defining functions with explicit operands and/or explicit results is discussed in a following section.

Function definition. Consider the following system input as an example of niladic function definition. The corresponding output sequence, as well as a detailed explanation, follows the example.

Example (system input):

```
DL NFAV
'NFAV CALCULATES THE AVERAGE OF A SET OF NUMBERS.'
VANUM IS EX NI
LAINP: GO LADO IF 0=RH VATST IS NI
VANUM IS VANUM, EX VATST
GO LAINP
LADO: 'THE NUMBERS ARE:'
VANUM
OB 'THE MEAN IS '
+/VANUM DV RH VANUM DL
NFAV
1 2 3 4
5 11
12 13 14 15

IB 64
```

Example continued (system output):

```
      DL NFAV
'NFAV CALCULATES THE AVERAGE OF A SET OF NUMBERS.'
VANUM IS EX NI
LAINP: GO LADO IF 0=RH VATST IS NI
VANUM IS VANUM, EX VATST
GO LAINP
LADO: 'THE NUMBERS ARE:'
VANUM
OB 'THE MEAN IS '
+/VANUM DV RH VANUM DL
      NFAV
NFAV CALCULATES THE AVERAGE OF A SET OF NUMBERS.
THE NUMBERS ARE:
 1.00000E00    2.00000E00    3.00000E00    4.00000E00
   5.00000E00    1.10000E01    1.20000E01    1.30000E01
   1.40000E01    1.50000E01
THE MEAN IS  8.00000E00
      IB 64
```

The first line switches the system from the execution mode to the function definition mode. This is accomplished by entering DL followed by the name that's to be assigned to the function. This APL statement is known as the header line of the function. The names of niladic functions must begin with NF and be followed by from one to three other alphanumeric characters. In the example above the function is named NFAV. An explanation of what the program does is provided by the message that is the first line in the body of the function definition.

Statements entered when the system is in the function definition mode are not executed at the time of entry. They are merely checked for syntax errors and then compiled and stored for later execution. The function definition mode is ended by entering DL after the last statement (either appended to the last statement, as in the example, or entered alone on the line following the last statement). When the function definition mode ends, the system returns to the execution mode. The entry following the function definition in the example above shows how to "execute" the function (by using its name alone on a line), and the succeeding lines show a sample "run" of the program.

Branching. In the normal course of events the execution of a defined function means the execution of the statements in the body of the function taken in order, starting with the first. This order may be altered by the "branch" function GO. This primitive function must have as its right argument the label of the line to which control is to be transferred. Labels must begin with LA and be followed by from one to three other alphanumeric characters. They must be the leftmost entry in the lines that are possible targets of the branch instruction, and they are separated by a colon from the lines they are labelling. A branch to the "null" label (LA alone, without any other characters) causes an exit from the program. This will of course also occur when the last line of the program is executed, provided it is not a branch.

The branch may be made conditional if the target label of the GO instruction is also used as the left argument of the primitive function IF. IF, in turn, must have as its right argument an APL expression that will evaluate to either "true" or "false", eg. 1 or 0. In the example above, there are instances of both the conditional and unconditional branches.

It is also possible to branch to one of a number of target statements by providing the GO function with an indexed vector of labels. In this case the index selects one specific label (unless the index is out of range, in which case control is transferred to the statement following the branch). An example would be

```
GO LAGO LA LAIN LA LAOUT [VAX]
```

where the variable VAX selects one of the five labels supplied.

Note: In APL-STARAN labels have no numeric value and may not be used as arguments for any primitive functions other than GO and IF.

Local and global variables. In the preceding example it is entirely possible that the variable VATST had been previously specified in the workspace and was holding some value that had to be used at a later time. In executing the function NFAV, however, its value was respecified, and after the function execution was completed VATST would be an empty vector, and its original value would have been lost. To prevent this from happening VATST may be specified as being local to the function. However, this will mean that during the execution of the function NFAV the global variable VATST will be unavailable for use. This will also be true of any functions invoked by NFAV. (In this particular example there are none, but there could have been).

The rule on local and global variables is that the system treats all variables as global unless they are made local to a function by appending their name, preceded by a semi-colon, to the name of the function when it is used in the header line of the function definition process. It is possible to make any number of variables local to a function by appending each successively to the header line of the function during the definition process. In each case they must be preceded by a semi-colon.

The first example below shows the effects on the variable VAX when it is left global and has its value respecified within a function. The second example shows how to save a global value by making the variable local to a function.

```
      VAX IS 2
      DL NFEX1
VAX IS 3 DL
      NFEX1
      VAX
 3.00000E00
      DL NFEX2;VAX
OB 'WITHIN THE FUNCTION VAX IS '
OT VAX IS 4 DL
      NFEX2
WITHIN THE FUNCTION VAX IS  4.00000E00
      VAX
 3.00000E00
```

Functions with explicit operands and/or results. It is possible to define functions with either one or two operands explicitly specified. The one operand specification is done by having a variable name specified in the function header line on the right side of the function name, and separated from it by a space. The names for such functions must begin with MF and be followed by from one to three other alphanumeric characters. In the following example VAX is an explicit operand variable, and the function MFSQR calculates the square root of this operand.

```
      DL MFSQR VAX
VAX * .5 DL
      MFSQR 10 4
 1.00000E00    1.41421E00    1.73205E00    2.00000E00
```

In this case the function must always be used with an operand on its right, or it will produce an error message. Local variables may also be specified following the right-hand operand; however, the operand variable(s) and the result variable (if there is one) are always local and should not be included among the local variable specification.

If two operands are specified (dyadic function) one must be on the left and one on the right, as is the case with all dyadic functions. Each operand must be separated from the function name by spaces, and the function name must begin with DF rather than MF. As an example, consider the following function which calculates the area of a triangle in terms of its base and height.

```
      DL VABAS DFTRI VAHGT
.5 TI VABAS TI VAHGT DL
      3 DFTRI 10
 1.50000E01
```

It is also possible to explicitly specify a result variable which is always local to the function involved, but permits a value to be returned as a result of the function invocation. This permits defined functions to be embeded within APL statements in much the same way that the basic arithmetic operations are used. A result variable is explicitly specified by entering its name followed by IS on the left of the function name (and left-hand operand if there is one). For example:

```
      DL VARES IS MFSQR VAX
VARES IS VAX * .5 DL
      10 + 3 TI MFSQR 4
 1.60000E01
```

All user defined functions may be used in other defined functions including their own definition, which allows recursive function definition.

None of an APL-STARAN function definition is compiled until the last line of the function has been read in. Compilation begins at this point and will stop when the first syntax error is encountered. The line number of the statement containing the error will be reported. The header line of the function is considered as line 0. If the error pertains to the use of a label in a branch statement which does not appear as a label on any line in the program, the line number of the error will be reported as the last line number plus one.

SECTION 5: APL-STARAN PERFORMANCE MEASUREMENTS

The execution times for various basic APL statements have been measured for both APL-STARAN and APL/370. The APL/370 version was run on the IBM System/370, Model 155 at Syracuse University in July and August 1975. During these tests the system load was light (between 4 and 6 users), but the 370 data will nevertheless show some small effects of core-swapping which will not be present in the STARAN data. In both cases the tests were run by a timing function MFTIM which caused 1000 executions of the test statement (except for the very long execution times which used a repetition factor of 200). The MFTIM function is shown below.

```
      DL VAR IS MFTIM VACNT;VATIM
VATIM IS IB 21
LAGO:NFTST
GO LAGO IF 0<VACNT IS VACNT-1
VAR IS (IB 21)-VATIM DL
      DL NFTST
```

The function NFTST was a one line niladic function which contained the test statement. A test was first run with NFTST being an empty function (i.e., only the header line), and the time for this execution was subtracted from all other measurements to correct for the overhead involved in running MFTIM. These times are listed in the columns labelled "Execution Time" of Table III. The times listed are in milliseconds and represent the CPU time required for a single execution of the test statement.

In all cases the results of the various primitive functions were assigned to variable VAX to avoid the overhead associated with result print-out. Even in this case there is an overhead associated with the assignment function. To correct for this two simple assignment tests were run, one for a scalar assignment (test 1) and one for a 256 element vector assignment (test 4). The columns labelled "Adjusted Times" have been corrected for this effect. They were obtained by subtracting the assignment overhead. The scalar overhead (test 1) was used to correct tests 2, 3, 11, 12, 16, and 17. The other tests were adjusted using the vector overhead.

Since the STARAN implementation uses essentially the same amount of time for a scalar as for a vector or higher dimensional array, the execution

time improvement of the STARAN over the 370 will be most evident for larger size vectors. The data clearly demonstrate this, yielding a factor of 4 or 5 as the average improvement for the case of a 256 element vector.

There appear to be two anomalies. First, the 370 uses an excessively large amount of time to compute the AN function (tests 12 and 13), but this is due to the need for a floating point conversion which is not required in the STARAN implementation. The second anomaly occurs in the plus reduction for the STARAN. This is due to the serial execution technique which is used, and will also be true for the times reduction. However it is not true for the AN, OR, MX, or MN reductions which use the full array capability of the STARAN.

TABLE III. COMPARATIVE PERFORMANCE MEASUREMENTS

Test Number	Test Statement	Pre-assigned Values	Execution Time		Adjusted Time	
			APL/370	STARAN	APL/370	STARAN
1	VAX IS 1.5		0.27	1.43	-	-
2	VAX IS VAA + VAB	VAA IS VAB IS -.3	0.80	4.33	0.53	2.9
3	VAX IS VAA TI VAB		0.73	4.33	0.46	2.9
4	VAX IS VAA	VAA IS VAB IS 256 RH -.3	1.53	2.0	-	-
5	VAX IS VAA + VAB		14.2	5.27	12.7	3.27
6	VAX IS VAA + .4		12.6	5.27	11.1	3.27
7	VAX IS VAA TI VAB		17.4	5.47	15.9	3.47
8	VAX IS VAA MN VAB		13.9	5.07	12.4	3.07
9	VAX IS VAA = VAB		17.2	5.03	15.7	3.03
10	VAX IS VAA > VAB		16.8	5.07	15.3	3.07
11	VAX IS MX/VAA		12.7	3.37	12.4	1.94
12	VAX IS AN/VAA	VAA IS 256 RH .5 DV .5	37.0	3.37	36.7	1.94
13	VAX IS VAA AN VAB	VAB IS VAA	53.2	4.07	51.7	2.07
14	VAX IS - VAA		8.1	5.10	6.6	3.10
15	VAX IS AB VAA		8.8	3.87	7.3	1.87
16	VAX IS VAA IO -1.5	VAA IS (255 RH - .1),-1.5	6.3	5.87	6.0	4.44
17	VAX IS +/VAA		13.6	378.7	13.3	377.3
18	VAX IS SG VAA		10.3	3.87	8.8	1.87

SECTION 6: APL-STARAN IMPLEMENTATION DETAILS

The AP module which runs the arrays is named SARREX (Staran Array Execution). It is written in APPLE and is loaded into Staran by the Goodyear system program SDM. It consists of a relocatable section which is loaded into pages 0, 1, and 2, plus a number of short absolute sections to be loaded at hexadecimal addresses 610, 8200, 8300, and B08A. There is also some residual debugging instrumentation located at absolute addresses 8400 and B030. This module is not designed to run under control of SDM but is rather initialized by a PDP-11 subroutine named INSTAR and then runs under control of a PDP-11 module named PARREX (PDP Array Execution Control).

After APL-STARAN has been initialized the AP comes to a halt by executing a WAIT instruction imbedded within the initialization routine. Subsequent calls on the AP occur when PARREX loads the necessary parameters into an argument area called APARG (AP Argument area - this area is a block of AP control memory which is shared with the PDP-11) and then issues an EXF (External Function Call) to start the AP. PARREX then continues with its own execution until it needs the AP again. At this point it waits on the completion of any previous AP call by looping with an EXF to test AP activity (all SARREX routines end with a WAIT instruction). PARREX then reloads APARG and the cycle repeats. Within SARREX the AP argument area is known as SAPARG.

The main SP (sequential processor) module is named APL. It is written in Macro-11 (assembler language for the PDP-11) and is linked with a number of subroutine modules (also written in Macro-11) by the LINK-11 system program. When loaded it occupies PDP-11 octal addresses from about 30000 through 137000. The addresses from 100000 through 137000 are shared with the Associative Processor starting at AP control memory address B000 (hexadecimal). These addresses contain the communications areas such as APARG. The other modules linked with APL are named PARREX, FARREX, SUPER, INSTAR, EXECFN, and DUMP. A number of subroutines from the Fortran System Library are also called by routines in FARREX and are incorporated into the load module when it is built by LINK-11. The PDP-11 transfer address is named START and is located at relocatable 0 in the APL module. This module is the master control for the entire APL-STARAN System.

AD-A054 943 SYRACUSE UNIV N Y F/G 9/2
LARGE SCALE INFORMATION SYSTEMS. VOLUME II.(U)
MAR 78 F30602-74-C-0335
UNCLASSIFIED RADC-TR-78-43-VOL-2 NL

3 OF 3
AD
A054943

END
DATE
FILMED
7-78
DDC

PARREX has 10 entry points named ARREX, LDATA, SDATA, OUTSIG, FIX, FRAFIX, FFLOAT, NUMFUZ, NUMCON, and $ERRA. All entries to PARREX come from APL as subroutine calls. When an error condition in the AP forces the discontinuance of a PARREX routine, the return to APL uses the ERROR entry in the APL module instead of the normal subroutine return. The various functions provided by the 10 entry points to PARREX are described in the documentation contained in the PARREX source code listing.

The FARREX module is a collection of 9 Fortran subroutines used by PARREX to perform certain sequential operations, such as the sum and product reductions. There are also a few array oriented operations namely, logarithm, exponential, floor and ceiling, which are not implemented as array operations in version 1.1 of APL-STARAN. In these cases PARREX uses SARREX to bring the data over to the PDP-11 where the calculations are performed serially by routines in the FARREX module.

SUPER is used by the APL module to handle the interface with the DOS-11 BATCH operating system. It has one entry point, SUPER, and the specific function requested is determined by a control code which must have been previously stored by the caller at SCNTRL. The entry is of the form of a subroutine call but the normal return sequence will be replaced by a jump to ERROR (in APL) if an abnormal condition occurs. Documentation found in the source code listing describe the specific functions provided by SUPER.

EXECFN is the APL-STARAN compiler, which has been written in APL-STARAN and then hand-translated into QCODE. The EXECFN module is the QCODE copy of this compiler. A source listing in APL may be found in the appendix to this report. In this source listing the functions QESC00, QESC02, QESM02, QESM04, QESM06, QAQESC, and QAQESC4 are qcode escape functions which are not available as ordinary APL-STARAN functions, but are used by the compiler to perform such tasks as symbol table manipulation, etc. The precise details of how these functions operate may be found in the source code of the main APL module.

The DUMP module is residual debugging instrumentation. Its entry point is at absolute octal address 130000, but there are no active references to that address. When the dump function is required it is necessary to patch the ERROR routine in the APL module to effect a jump to the dump entry point. This may easily be accomplished by replacing the DOS command $RUN

DK1:APL with a $GET, $MODIFY, $BEGIN sequence. The effects of a call to DUMP will be to create a Multics file APL.DMP which will contain most of the data in the APL-STARAN system tables and communications areas. APL-STARAN will then be terminated, the AP will be cleaned up, and control will be returned to the DOS-11 Operating System (unless, of course, the bug has destroyed the dump routine).

```
     ∇ VARES←EX VAIC
[1]    VARES←VAFN←CΔEMPT
[2]    VACP←¯1+VAOR←IB 48
[3]    VAIC←95⌊VAIC,3
[4]   LASKIP:→LASKIP IF 32=VAIC[VACP←VACP+1]
[5]    →LAGO IF 68≠VAIC[VACP]
[6]    →LAGO IF 76≠VAIC[VACP+1]
[7]    VAIC[VAQ←VAOR+¯1+⍴VAIC]←13
[8]    →LAFEND
[9]   LAFIN:VAQ←VAOR+¯1+⍴VAIC←VAIC,(95⌊OT NI),13
[10] LAFEND:→LAFEND IF 32=VAIC[VAQ←VAQ-1]
[11]   →LAFIN IF 76≠VAIC[VAQ]
[12]   →LAFIN IF 68≠VAIC[VAQ-1]
[13]   VAIC[VAQ+1]←3
[14]   VAFN←0
[15] LAGO:VACP←VACP-1
[16] LATRAN:→LAPUT IF 8<VAQ←CΔPTOQ[VAOR+VAIC[VACP←VACP+1]]
[17]   →(LAC0,LAC1,LAC1,LAC3,LAC4,LATRAN,LAC6,LAC7,LAC8)[VAOR+VAQ
       ]
[18] LAC3:→LAC1 IF 1=CΔPTOQ[VAOR+VAIC[VACP+1]]
[19]   VAQ←14
[20] LAPUT:VARES←VARES,VAQ
[21]   →LATRAN IF 17≠VAQ
[22]   QESM06 1
[23]   →LATRAN
[24] LAC0:→LAPUT IF VAZ←8<VAQ←CΔMTOQ[CΔPTOM⍳VAIC[VAW←VACP←VACP+1
       ]+256×VAIC[VACP]]
[25]   →LAC0GO IF 5>VAQ
[26]   →LAC8 IF 8=VAQ
[27] LAC6:QAQESC4
[28] LAC0LP:VAZ←(38×VAZ)+VAX←64-43×VAY
[29] LAC0GO:→LAC0LP IF 1≥VAY←CΔPTOQ[VAOR+VAX←VAIC[VACP←VACP+1]]
[30]   →LAC6 IF ¯3>VAW-VACP←VACP-1
[31]   →LAC0SM IF VAQ≠4
[32]   VARES←VARES,(VAZ+208),VAQ
[33]   →LATRAN
[34] LAC0SM:→LAC6 IF VAW=VACP
[35]   →LAC0IN IF(VAY←⍴VAΔSYM)>VAX←(VAΔSYM⍳VAZ←VAZ+VAQ×54872)-
       VAOR
[36]   VAΔSYM←VAΔSYM,(((VAX←QESM02 CΔTYPE[VAOR+VAQ])-VAY)⍴0),VAZ
[37] LAC0IN:VARES←VARES,(CΔSMSTRT-16×VAX),VAQ
[38]   →LATRAN
[39] LAC1:VAQ←CΔEMPT
[40] LAC1GO:VAV←VACP←VACP-VAY←VAP←1
[41]   VAZ←54872
[42]   VAX←0
```

```
[43] LAC1LP:→(LAC1AL,LAC1NO,LAC1SG,LAC1DP,LAC1GN)[VAOR+
      4⌊CΔPTOQ[VAOR+VAW←VAIC[VACP←VACP+1]]]
[44] LAC1NO:VAX←(10×VAX)+VAW-48
[45]  →LAC1LP
[46] LAC1SG:→LAC1LP IF VAV=VACP+VAY←¯1
[47]  →LAC6
[48] LAC1DP:→LAC1LP IF VACP=VAZ←VACP+VAZ-54872
[49]  →LAC6
[50] LAC1AL:→LAC1GN IF 69≠VAW
[51]  →LAC1NE IF 94=VAP←VAIC[VACP+VAW←1]
[52]  →(LAC1GN,LAC1EX)[VAOR+1=CΔPTOQ[VAOR+VAP]]
[53] LAC1NE:→LAC6 IF 1≠CΔPTOQ[VAOR+VAP←VAIC[1+VACP←VACP-VAW←
      ¯1]]
[54]  VAZ←VAZ+1
[55] LAC1EX:VAP←VAP-48
[56]  →LAC6 IF 46=VAR←VAIC[VACP←VACP+2]
[57]  →LAC1GE IF 1≠CΔPTOQ[VAOR+VAR]
[58]  VAZ←VAZ+1
[59]  VAP←(10×VAP)+VAR-48
[60]  →LAC6 IF(3=VAR)∨1=VAR←CΔPTOQ[VAOR+VAIC[VACP←VACP+1]]
[61] LAC1GE:VAP←10*VAW×VAP
[62]  VAZ←VAZ+2
[63] LAC1GN:→LAC6 IF(¯1=VAX+VAY)∨¯3=VAV+VAZ-2×VACP
[64]  VAQ←VAQ,VAY×VAP×VAX×10*0⌊VAZ-VACP←VACP-1
[65] LAC1NX:→(LAC1ND,LAC1GO,LAC1GO,LAC1GO,LAC6,LAC1NX,LAC6,
      LAC1ND)[VAOR+7⌊CΔPTOQ[VAOR+VAIC[VACP←VACP+1]]]
[66] LAC1ND:→LAC1EN IF 1≠ρVAQ
[67]  VAQ←CΔEMPTρVAQ
[68] LAC1EN:VARES←VARES,(VAFN QESC00 VAQ),5
[69]  VACP←VACP-1
[70]  →LATRAN
[71] LAC4:VAQ←CΔEMPT
[72] LAC4LP:→LAC4QT IF 39=VAX←VAIC[VACP←VACP+1]
[73]  →LAC6 IF 3=VAX
[74] LAC4GC:VAQ←VAQ,VAX
[75]  →LAC4LP
[76] LAC4QT:→LAC4GC IF 39=VAIC[VACP←VACP+1]
[77]  VAQ←VAQ,CΔEMPT
[78]  →LAC1ND
[79] LAC8:→LAPUT IF 17=VAQ←CΔPTOQ[VAOR+VAIC[VACP←VACP+1]]
[80]  →LAC8 IF 7≠VAQ
[81] LAC7:VACP←VAOR+ρVAIC←6,7,VARES
[82]  VAA←VAB←VAL←VARES←CΔEMPT
[83]  VAV←8+VAZ←VAOR
[84]  VAW←VAX←VAY←8ρVAJ←0
[85]  →LASLP IF 0=ρVAFN
[86]  VAP←VAA←VAB←0
[87]  →LAC6 IF(ρVAIC)=(VACP←VAIC⍳17)-VAOR
[88]  VAIC[VACP]←16
[89]  QESM06 0
```

```
[90] LALOCL:→LALCND IF 26≠VAIC[VACP←VACP-3]
[91]  →LAC6 IF 0≠VAIC[VACP+2]
[92]  VAL←VAL,VAIC[VACP+1]
[93]  →LALOCL
[94] LALCND:→LAFNAM IF 0≠VAIC[VACP+2]
[95]  VAB←VAIC[3+VACP←VACP-2]
[96] LAFNAM:→LAC6 IF(0=VAQ)∨3<VAQ←VAIC[VACP+2]
[97]  VAFN←,VAIC[VACP+1]
[98]  →LANOLA IF 0≠VAIC[VACP]
[99]  VAA←VAIC[1+VACP←VACP-2]
[100]LANOLA:→LANORS IF 21≠VAIC[VACP]
[101] →LAC6 IF 0≠VAIC[VACP-1]
[102] VAP←VAIC[1+VACP←VACP-3]
[103]LANORS:→LAC6 IF(VAQ≠1+(×VAA)+×VAB)∨(VACP≠VAOR+2)∨26≠VAIC[
      VACP]
[104] VARES←136,VAA,VAB,VAP,VAL,0
[105] VAA←VAB←0⍴VAL←,208
[106] VAJ←1
[107] QESM06 1
[108] →LASLP IF(VAOR+⍴VAIC)≠VACP←VAIC⍳17
[109] →LAC6 IF 28≠VAIC[VACP←VACP-1]
[110] →LAFOUT IF 16=VAIC[VACP-1]
[111] VAIC←VAIC,17,28
[112]LASLP:VAR←⍴VARES←VARES,((~VAJ)⍴131),17
[113] →LAC6 IF VAJ←(VAP=0)∨(VAP≠25)∧6<VAP←CLQTOC[VAOR+VAIC[VACP←
      VACP-1]]
[114]LASYN:VAQ←VAP
[115]LANEXT:→LANEXT IF 208≤VAP←VAIC[VACP←VACP-1]
[116] VAP←CΔQTOC[VAOR+VAP]
[117] →(LAA,LAB,LAC,LAD,LAE,LAD,LAG,LAD,LAI,LAJ,LAK,LAL,LAM,LAN,
      LAO,LAP,LAQ,LAR,LAS,LAT,LAU,LAV,LAW,LAX,LAY,LAZ,LAC6,LAC6)
      [VAOR+VAQ]
[118]LAA:→LAAOP IF 15=VAIC[VACP]
[119] →LAC6 IF(11≠VAP)∧13≠VAP
[120] VARES←VARES,14,VAIC[VACP]
[121] VAIC[VACP]←20
[122]LAAOP:VAP←CΔQTOC[VAOR+VAIC[VACP←VACP-1]]
[123] →LAM
[124]LAB:VARES←VARES, 12 24 204 132
[125] →LADH
[126]LAC:VARES←VARES,13
[127]LAD:VARES←VARES,VAIC[VACP+1]
[128]LADH:→LASYN IF(6≤VAP)∧24≥VAP
[129] →LAC6
[130]LAE:VAY[VAZ]←VAY[VAZ]+1
[131]LAM:VARES←VARES,VAIC[VACP+1]
[132]LAEM:→LASYN IF(1≤VAP)∧6≥VAP
[133] →LAC6
[134]LAG:→LAC6 IF VAV<VAZ←VAZ+1
[135] VAV[VAZ]←VAX[VAZ]←VAOR+⍴VARES
```

```
[136] →LAGEX IF(1≤VAP)∧6≥VAP
[137] →LAC6 IF(19≠VAP)∧20≠VAP
[138] VARES←VARES,9,24
[139] →LASYN
[140]LAGEX:VARES←VARES,10
[141] →LASYN
[142]LAI:→LAD IF 0≤VAY[VAZ]←VAY[VAZ]-1
[143] →LAC6
[144]LAJ:VARES←VARES,30
[145] →LAD
[146]LAK:VARES←VARES,28,VAIC[VACP+1],29
[147] →LAEM
[148]LAL:VARES←VARES,VAIC[VACP+1]
[149] →LASYN IF 6≥VAP
[150] →LAC6
[151]LAQ:→LAC6 IF(VAQ←134=VARES[VAOR+¯1+ρVARES])∧6<VAP
[152] →LARQ IF VAQ∨6<VAP
[153] →LARQNI
[154]LAR:→LARQ IF 133=VARES[VAOR+¯1+ρVARES]
[155]LARQNI:VARES←VARES,135
[156]LARQ:→LAC6 IF VAP=0
[157]LAN:VARES←VARES,VAIC[VACP+1]+128×6<VAP
[158] →LASYN IF 24≥VAP
[159] →LAC6
[160]LAO:→LAN IF 0≠VAP
[161] →LAC6
[162]LAP:VARES←VARES,VAIC[VACP+1]
[163] →LASYN IF(5=VAP)∨6=VAP
[164] →LAC6
[165]LAS:→LASTST IF 133=VARES[VAOR+¯1+ρVARES]
[166] VARES←VARES,135
[167]LASTST:→LAM IF(11≠VAP)∧13≠VAP
[168] VARES←VARES,VAIC[VACP+1]-62
[169] VAP←CΔQTOC[VAOR+VAIC[VACP←VACP-1]]
[170] →LAD
[171]LAT:→LAC6 IF VAW[VAZ]←0≠VAY[VAZ]
[172] →LATEX IF(1≤VAP)∧6≥VAP
[173] →LAC6 IF(19≠VAP)∧20≠VAP
[174] VARES←VARES,25,24
[175] →LASYN
[176]LATEX:VARES←VARES,26
[177] →LASYN
[178]LAU:→LAC6 IF(VAOR≥VAZ)∨0≠VAY[VAZ]
[179] →LAURS IF(VAP=17)∨VAP=18
[180] →LAUNFN IF VAP≠16
[181] VAIC[VACP+1]←VAIC[VACP+1]+1
[182]LAURS:VARES←VARES,VAIC[VACP+1]+111
[183] →LAC6 IF(VAW[VAZ]=0)∨VARES[VAX[VAZ]]=9
[184] VARES[VAX[VAZ]]←8
[185] →LAUOK IF(18=CΔQTOC[VAOR+208⌊VARES[VAQ-1]])∨128≤VARES[VAQ←
      VAX[VAZ]-1]
```

```
[186] →LAUOK IF 26=CΔQTOC[VAOR+208⌊VARES[VAQ]←VARES[VAQ]+
      128]
[187] →LAC6
[188]LAUNFN:→LAUZ IF 25=VAP
[189] →LAC6 IF(0=VAP)∨5<VAP
[190] VARES←VARES,VAIC[VACP+1]+VAQ←21=VARES[VAX[VAZ]-1]
[191] →LAC6 IF(18=CΔQTOC[VAOR+208⌊VARES[VAQ-1]])∨(128≤VARES[VAQ←
      VAX[VAZ]-1])∨VAQ∧VAP≠5
[192]LAUOK:VAZ←VAZ-1
[193] →LASYN
[194]LAUZ:→LAC6 IF(VAW[VAZ]=0)∨(VARES[VAX[VAZ]]=9)∨VARES[VAX[VAZ
      ]-1]≠17
[195] VARES[VAX[VAZ]-1,0]←16,17
[196] VAQ←ρVARES←VARES, 130 0
[197]LAUZLP:VAB←VAB,ρVARES←VARES,VAIC[1+VACP←VACP-2]
[198] →LAUZLP IF 25=CΔQTOC[VAOR+VAIC[VACP]]
[199] VARES[VAOR+VAQ-1]←(ρVARES)-VAQ
[200] VAZ←VAZ-1
[201] →LAZGO
[202]LAV:→LAC6 IF 25≠VAP
[203] VARES←VARES,129
[204] VACP←VACP-2
[205] →LAZPUT
[206]LAW:→LAC6 IF(16≠VAIC[VACP-2])∨(25≠VAP)∨(VAOR+ρVAL)≠VAL⌽VAQ←
      VAIC[VACP-1]
[207] VAL←VAL,VAQ
[208] VAA←VAA,VAR
[209]LAX:→LAC6 IF(0=ρVAFN)∨(VAZ≠VAOR)∨VAY[VAOR]≠0
[210] QESM06 1
[211] →LANOLD IF VAJ∨(21=VAP)∨(23=VAP)∨(78=VAP)∨(79=VAP←VARES[
      VAQ-1])∨(206=VAP)∨207=VAP←VARES[VAQ←VAOR+¯1+ρVARES]
[212] VARES←VARES,139
[213]LANOLD:VAIC[VAIC⌽17]←16
[214] →LASLP IF(VAOR+ρVAIC)≠VACP←VAIC⌽17
[215] →LAC6 IF 28≠VAIC[VACP←VACP-1]
[216] VAIC←VAIC,17,28
[217] →LASLP IF 16≠VAIC[VACP-1]
[218] VAA←(1+ρVARES←VARES,(~VAJ)ρ131),VAA
[219] →LAFOUT IF 0=ρVAB
[220] VAV←VAOR+ρVAIC←VAOR+VAB-1
[221] VAQ←VAOR+ρVAL
[222]LAINBR:→LAC6 IF VAQ=VAP←VAL⌽VARES[VACP←VAIC[VAZ]]
[223] VARES[VACP]←2×VAA[VAP]-VAB[VAZ]
[224] →LAINBR IF VAV>VAZ←VAZ+1
[225]LAFOUT:→VAFN QESC02 VARES
[226]LAY:→LAC6 IF(VAZ≠VAOR)∨VAY[VAOR]≠0
[227] →QESM04 VARES
[228]LAZ:VARES[VAOR+¯1+ρVARES]←128
[229]LAZPUT:VAB←VAB,ρVARES←VARES,VAIC[1⌽VACP]
[230]LAZGO:→LASYN IF VAJ←((22=VAP)∨23=VAP←CΔQTOC[VAOR+VAIC[VACP←
      VACP-1]])∧128=VAIC[VACP]
[231] →LAC6  ∇
```

References

[1] APL/360-OS and APL/360-DOS User's Manual, IBM Publication GH20-0906 (1970).

[2] Sandra Pakin, APL/360 Reference Manual, Science Research Associates, Inc., (1972), 192 pp.

[3] Kenneth J. Thurber and John W. Myrna, "System Design of a Cellular APL Computer", IEEE Transactions on Computers (April, 1970), Vol. C-19, No. 4, pp. 199-212.

[4] A. Hassitt, J. W. Lageschulte, and L. E. Lyon, "Implementation of a High Level Language Machine", Communications of the ACM (April, 1973), Vol. 16, No. 4, pp. 291-303.

[5] A. D. Falkoff and K. E. Iverson, "The Design of APL", IBM J. Res. Devel. (July, 1973), Vol. 17, pp. 324-334.

[6] Tom McMurchie, "A Limited Character APL Symbolism", APL Quote-Quad (Nov., 1970), Vol. 2, No. 4, pp. 3-4.

[7] P. E. Hagerty, "An APL Symbol Set for Model 35 Teletypes", APL Quote-Quad (Sept., 1970), Vol. 2, No. 3, pp. 6-8.

[8] Glenn Seeds, "APL Character Mnemonics", APL Quote-Quad (Fall, 1974), Vol. 5, No. 2, pp. 3-9.

[9] A. D. Falkoff and K. E. Iverson, APLSV User's Manual, IBM Publication SH20-1460 (1973).

[10] R. A. Kelly, "APLGOL, An Experimental Structured Programming Language", IBM J. Res. Devel. (Jan., 1973), Vol. 17, pp. 69-73.

[11] M. A. Jenkins, A Control Structure Extension to APL, Department of Computing and Information Science, Queen's University, Kingston, Ont., Technical Report No. 21, (Sept., 1973), 13 pp.

[12] James R. Bell, "Threaded Code", Communications of the ACM (June, 1973), Vol. 16, No. 6, pp. 370-372.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.